2010

Zackup, a scalable centralized backup service

Benjamin Allen

R·I·T

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Zackup, A Scalable Centralized Backup Service

Benjamin S. Allen

Master of Science in Networking and Systems Administration

Rochester Institute of Technology
B. Thomas Golisano College of Computing and Information Sciences
Master of Science in Networking and Systems Administration

Thesis Approval Form

Title: **Zackup, A Scalable Centralized Backup Service**
Author: Benjamin S. Allen

Thesis Area: Systems Administration

Professor Bill Stackpole, Committee Chair

Signed: _____ Date: _____

Professor Luther Troell, Committee Member

Signed: _____ Date: _____

Professor Bo Yuan, Committee Member

Signed: _____ Date: _____

Abstract

The currently available open source, centralized to disk, backup services use a custom data storage structure to effectively store backup sets. While custom data storage structures are necessary to store backup sets in differential or space efficient manner, they often lead to hard to navigate structures and increase the complexity of the backup service. To solve these problems a backup service was developed that relies on the Sun Microsystems, Inc. open sourced file system ZFS. With the variety of features ZFS includes, the majority of common tasks to create backup sets were delegated to the file system. Along with ZFS, a number of Ruby libraries were used to help further reduce the amount of author created code. Lastly to increase the scalability of the developed backup service a job queuing based communication architecture between service entities was employed allowing for a multiple backup node solution.

Contents

List of Figures

1

Introduction

The main objective of this thesis was to develop a centralized backup service that takes advantage of the ZFS filesystem. ZFS is a Sun Microsystems open source file system and offers the ability to create snapshots of a filesystem. A snapshot allows for the filesystem itself to create a point-in-time pointer to its files. Using ZFS eliminated the need to create the code for creation of backup sets. A scheduling daemon, web front-end, in addition to a backup daemon were developed to take advantage of ZFS. Zackup contains a reduced amount of author created code than a comparable service with author implemented backup sets. Lastly, Zackup is developed and released open source, and is free to the public under the MIT license.

The secondary objective of this thesis was to study and implement a proper backup set retention policy that most efficiently takes advantage of ZFS' capabilities. Retention policy methods that were considered: "keep at least N versions", "keep no more than N versions", "after N time units expire", and lastly "do not expire before N time units have passed". Moreover, strategies consider the possibility of a user requesting certain files never to be deleted.

A tertiary objective of this project was to expand the backup daemon to be scalable across multiple servers. The main point of this objective was to improve performance of backups by utilizing many local fast storage pools and by dividing the client load. To allow for many backup daemons a job queueing architecture were employed. This architecture features storage of job status and data in a centralized database, where all backup daemons as well as the web front-end and scheduling daemon are able to access and modify jobs as needed. Each distributed backup daemon polls the queue when it is ready to process new jobs. Having all entities communicate in this fashion allows for a scalable backup service.

2

Prior Work Review

2.1 Introduction

Various existing works were reviewed and influenced the direction and goals of Zackup. A number of literary works were reviewed for recent research in the arena of backup. BackupPC, an open sourced software, in which the author has run in production for two or more years, was the spark that inspired Zackup. Additional influence was gathered from examining BackupPC and other existing backup solutions.

2.2 Literary Review

The following is a review of four literary works that influenced the direction of Zackup. Each work was evaluated for valuable information pertaining to related concepts and ideas to Zackup's goals.

The authors of "LifeBoat - An Autonomic Backup and Restore" created a product called LifeBoat that aims to do local and remote backups, as well as bare metal restores of systems that have the Windows operating system installed [1]. They found the Windows operating system presents some particularly difficult problems during both backup and restore [1]. The authors had to store additional metadata from the operating system to make bare metal backup and restoration possible.

Zackup's design limited the practicality of supporting bare metal restoration. This would have required a tighter integration with the clients operating system as found by LifeBoat's authors. Typically tighter integration requires a custom created standalone client that lives on the client system. Using a custom client would greatly expand the complexity of supporting multiple client platforms.

In "Logical vs. Physical File System Backup", backup strategies were compared. Logical backups are where entire files are written to the backup media,

and physical backups are where blocks are directly copied to the backup media. The authors looked at the advantages and disadvantages in terms of performance and throughput [4]. Physical backup and restore achieved much higher throughput than logical backup and restore [4]. The reason for this is that a logical backup and restore has to create related file metadata for each single file; in addition, the fairly random order in which files exist on a disk further reduces performance. A physical backup reads a disk sequentially block to block, and as a result can be performed faster.

The design of Zackup uses logical backups. The reason for this is cross system compatibility. Zackup takes advantage of a tool named RSYNC to get some of the benefits of a physical backup like differential copy. Zackup's objectives did not include finding the best technique of transferring backups, rather creating a lightweight backup service by taking advantage of already existing software packages and libraries. File transferring techniques evaluated that require full copy of files transferred across the network were judged to be too costly to performance to be use in Zackup's design.

"Surviving Internet Catastrophes" tries to solve the problem of protecting data. The solution put forth takes advantage of the fact that most vulnerabilities that are found and exploited, especially worm type malware, typically

only exploit one piece of software or operating system. As such, the designed backup infrastructure relies on multiple operating systems and multiple applications. The design relies on the most heterogeneous environment possible. A backup system called Phoenix Recovery Service was created to manage this heterogeneous environment [5]. Phoenix Recovery Service relies on various heuristics to help ensure that each piece of data is stored across many locations, operating systems, and applications that statistically ensure the data to be safe. Their design has a paradox built into it however. Since their design relies on heterogeneity, installing their common backup software on all the systems breaks that requirement. It was found that their heuristics allowed for over 0.99 probability that user data survives attacks of single and double-exploit pathogens [5].

Zackup's objectives are more focused on creating a fast and reliable service, and not focused on how to survive an internet catastrophe. However, as a result of the third goal of the project of being scalable, additional redundancy can be built into this service by adding additional backup nodes thus increasing its survivability of various malware and attack.

In "A Cooperative Internet Backup Scheme" the authors designed and implemented a novel peer-to-peer backup technique that uses internet peers as

backup servers. In return for storing your data on a peer, you must store some of your peers data on your machine. In other words, it is cooperative [6]. A cooperative scheme is vulnerable to various attacks. The simplest attack is a user backing up their data, but not storing any of their peers data. To solve this, the authors implemented random challenges against peers to ensure that the data is still available and actively being stored. With most of the problems solved, the authors found their scheme is roughly one to two orders of magnitude less expensive than existing internet based backup services [6].

Zackup is designed as a centralized service, not distributed. The above authors' project is the opposite of Zackup's design. Backups should live in a very well protected environment, and not be trusted to unknown peers for safe keeping. Peers have no motivation to keep other peers' data other than simply receiving backup space for themselves. Since backups traditionally are to help guarantee safe keeping and subsequent recovery of data, adding another variable of uncertainty is contradictory.

2.3 Similar Backup Services Review

Hereunder, three examples of backup systems with similar characteristics as Zackup are described and compared against Zackup's design. The software investigated was Apple Inc's Time Machine, and two open source projects

BackupPC and Zetaback. This information was gathered by the author personally investigating these pieces of software unless otherwise cited.

2.3.1 Zetaback

Zetaback is a "thin-agent based ZFS backup tool" [7]. Running from a centralized location, with a client software installed on each client, Zetaback backs up ZFS filesystems on the client using ZFS Send and ZFS Receive [7]. These functions are similar to the unix DD utility, where ZFS Send streams the specified filesystem to a given output, typically STDOUT or a file. ZFS Receive listens for this stream typically on STDIN. These functions used together can copy entire ZFS filesystems. With the addition of SSH or similar protocol which allows data to be piped or tunneled, filesystems can be copied over the network. Zetaback takes advantage of these functions to perform backups of ZFS based clients. It is designed specifically to be used with ZFS as a requirement for the client and server systems, and would be an excellent tool to backup servers or other machines in a homogeneous ZFS environment. However since ZFS is not ubiquitous, Zetaback has limited usefulness in heterogeneous environments.

As a feature included in Zackup's future work, Zetaback's ability to backup ZFS filesystems will be useful to allow for bare-metal restorations of ZFS

based systems. Bare-metal restorations refer to the ability of a backup service to restore an entire operating system, installed applications, and user data back to a working state. Where as Zackup's current methods of backup does not offer a reliable way to backup and restore entire systems. Zackup currently only guarantees the ability for specific directories and files to be backed up and does not guarantee file and directory metadata. With an entire ZFS volume backed up similar to Zetaback, a system would be able to be booted off a rescue disc, and it's filesystem fully restored to the latest backup. As Zackup is using ZFS for is backend storage, the integration of backing up ZFS filesystems with the ZFS Send and Receive facilities will be possible in the future.

To solve Zetaback's limitation of requiring clients to be running the ZFS filesystem, a more generalized method of file transfer, RSYNC over SSH, was chosen for Zackup. SSH and the RSYNC utility can be installed on the client without the potential drastic changes that are required to migrate to a new filesystem. The ease of adoption of RSYNC over SSH will lead to Zackup being more useful in a heterogeneous environment.

2.3.2 Apple Time Machine

Apple's Time Machine backup software has been included in Mac OS X 10.5 "Leopard" and 10.6 "Snow Leopard" operating systems. It is a closed source propriety backup software. Time Machine has two basic modes of operation. First backing up to a locally connected external hard drive, and second backing up to a network connected Apple Time Capsule device or Apple Airport Extreme with attached external hard drive. The primary difference between the two modes is the use of sparse disk images. A sparse disk image is a file that can be mounted by Mac OS X as a virtual disk. The file grows as data is written to the mounted disk image. A sparse disk image is used when Time Machine is used in conjunction with network connected storage. Other than this difference, the following backup strategy is used in both modes. Apple's Time Machine initially creates a full copy, minus exclusions, of the backed up hard drive. Time Machine uses file copies, not block copies, so each time a file is changed in subsequent backups, the entire file is copied to the backup location. At the backup location Time Machine will never overwrite existing files, rather it will create a new directory for each new backup set. Time Machine runs a backup every hour its backup media is available to the system. To conserve disk space, Time Machine merges each

days worth of backups keeping the latest copies of all files. In addition, a weeks worth of daily backups are merged into one weekly backup. Weekly backups are kept until the backup location runs low on disk space. This retention policy is not configurable. Between backup sets, identical files are stored only once. Additional references to duplicate files are made with hard links reducing storage requirements.

Apple's Time Machine offers a few methods of restoration that are unique. First a somewhat gimmicky graphical interface that lets you visually "travel" back in time for each Finder window that is open. This graphical interface allows for an easy method to restore specific files or directories. Second, Mac OS X's installer allows the user to restore an entire system from a Time Machine backup. As a result if a system has a failed hard drive, the user can replace the hard drive, boot the Mac OS X install DVD and restore the entire Mac OS X install, applications, and data from the backup.

While Time Machine offers an easy to use and reliable method of backup, it relies mainly on that the backup media is relatively close in physical proximity. In the case of using a directly connected external drive, the backup media is literally on the same desk as the computer being backed up. In the case of using network based storage, a consumer device, the Apple Airport

Extreme or Apple Time Capsule must be used. There is the ability to enable the use of unsupported network storage such of NFS or Samba shares if the user is willing to execute a command on the command line interface. Apple does not support or condone the use of NFS, Samba, or other normally not shown network mounts. While Time Machine is a great utility for the small office or home user, it does not seem suited or designed for larger scale use. In addition, since Time Machine runs on the client, it relies on the user to be responsible. The user is typically capable of disabling backups. If used in an enterprise environment additional policy enforcement would need to be developed on top of Mac OS X and Time Machine to ensure backups were continually enabled and occurring. Moreover, some sort of load balancing mechanism where clients would be grouped to specific network backup media would be needed.

Zackup attempts to solve the enterprise scale problems facing Time Machine. First it takes a lot of the control away from the day to day users' hands. Backup jobs are initiated from a centralized service. Based on the setup schedule, Zackup pulls files from the client. This is instead of the client pushing files to the backup storage. The centralized approach allows for the enterprise administrator to more closely monitor and ensure back-

ups are actually occurring. In addition, as Zackup's Backup Daemon can scale across multiple servers or storage pools, clients can be effectively load balanced across backup resources. Since Zackup's presence on the client is limited to the installation and running of an SSH daemon and the RSYNC utility, no additional policy enforcement measures are needed. If for example the RSYNC utility is uninstalled by the user, the backup job will fail with an error. The administrator or user would know what the error stated and would be able to correct the problem.

Zackup can live on any Solaris of FreeBSD installed hardware. It does not rely on consumer based hardware like Time Machine. As a result more reliable and capable hardware can be used. Time Machine does not have a configurable backup schedule or retention policy. This forces the user to accept Apple's configuration on when a backup should occur and how long their data should be retained. As all users have separate and distinct risk tolerances, as well as certain data may be more important than other data, Zackup allows for all combinations of schedule, retention policy, and specific configuration of what data is to be backed up. Lastly, the Time Machine software only runs on Mac OS X and is closed source. As many enterprises or even homes have a heterogeneous environment of Microsoft Windows, Linux,

and similar, Time Machine will only serve a portion of the clients needing backup.

One area that Zackup does not do as well as Time Machine is bare metal restorations. With the integration built into the Apple OS X installer for full system restorations, Time Machine backups are an attractive way for the typical Mac OS X to do backup. Zackup does not have the integration necessary to be able to restore an entire operating system, its applications, and data. Primarily Zackup is capable of backing up and restoring data, with some limited ability to backup and restore applications. Time Machine has the luxury of only being used on a single closed platform, so Apple's developers could more easily ensure its ability to perform any additional tasks needed to successfully restore an entire system.

2.3.3 BackupPC

BackupPC, available at http://backuppc.sourceforge.net, is centralized in design, open source, includes compression, file de-duplication, and utilizes RSYNC and Samba. RSYNC is a utility for syncing remote and local copies of the same directories and files. It vastly speeds up the syncing process by only transferring the differences in files instead of the entire file. Identical files in BackupPC between backup sets are reduced, similar

to Time Machine, with the use of hard links. BackupPC goes one step further and de-duplicates backed up files. It performs de-duplication on files not based on filename or directory location, but based on the a file's MD5 hash. As a result, identical files with different filenames and different paths are de-duplicated. BackupPC also compresses files with Gzip compression. Because of the GZIP compression however, BackupPC's backup storage is not browsable at the file system level. If the BackupPC service fails and is not restorable, it would take a tremendous amount of time and effort for an administrator to locate specific files from the existing backup sets. BackupPC is a centralized service, based on that backups are initiated on the server and files are transferred by pulling from clients. However, BackupPC can only exist on a single server. Its design does not allow for its various software entities to be split up horizontally across multiple backup resources. As such the only way to increase capacity for a BackupPC installation is to add resources, or scale vertically. Alternatively distinct installations could be created of BackupPC on separate servers. As a result clients would need to be assigned to specific BackupPC installations. This model of load balancing is limited, as certain clients will have unexpected backup storage requirement growth or conversely will require less backup storage. Thus this may result in unequal load on the backup resources.

Zackup borrows a number of features from BackupPC. RSYNC over SSH is used for transferring files from the client. ZFS enables both compression and incremental backups. ZFS block level duplication is currently under development for ZFS and can be independently integrated in the future. Block level de-duplication will go one step further than what BackupPC is currently capable of, and will actually remove individual blocks of files that are duplicate across the filesystem, where BackupPC will only remove duplicate whole files.

One area where Zackup solves a large issue of BackupPC, is the ability to have filesystem level browsable backup sets. Being able to browse the backup sets directly is important incase of the worse case scenario happens and the backup service fails and is not restorable. In this case an administrator of Zackup can easily identify what ZFS filesystem belongs to a certain client, at what date and time each individual backup set was created, and create a restore archive manually of the needed files and directories. All filesystem names are named by the clients IP address. Backup sets are named as the date and time they were made. If the server fails thats running Zackup, and the backup storage disks are still usable, the disks can be installed into any system with the capable storage interconnects (i.e. SATA, SCSI, FiberChannel, etc),

and import the ZFS filesystems that were used for the service. No specific hard drive controller is needed unless a specific hardware RAID was used on the old server.

BackupPC is deployed on a single server, and is not capable of effectively load balancing across multiple servers. Zackup solves this scalability issue by using a job queueing architecture, where each software entity either creates or consumes jobs. As a result, separate instances of the Backup Daemon can be run on multiple servers. Each Backup Daemon can then access separate storage, use separate network connections, and have their own hardware resources. Clients are then load balanced across the Backup Daemons to effectively scale the service.

BackupPC currently supports the use of a Samba share or CIFS protocol to transfer files from the client. This transfer protocol allows for out of the box support of Microsoft Windows clients. Moreover, BackupPC supports the use of RSYNC alone. RSYNC has a daemon component that can be run separately from SSH. While this feature is not quite as important as Samba support, it does give the user more file transfer options. Currently, Zackup only supports RSYNC over SSH. As a result, the support of Windows clients is limited to inelegant client software installs. Noted in the future work

section, Zackup needs the ability to easily backup Windows clients.

2.4 Summary

A number of prior works have influenced the direction of Zackup. The sparking idea for Zackup was generated from BackupPC. The general architecture, of centralized backup using RSYNC over SSH, was taken from BackupPC and adapted to be used with ZFS as the storage backend. The other works discussed were studied to find if their architecture or designs should be used over Zackup's initial design ideas. It was found that sticking with a design similar to BackupPC satisfied the goals of this project.

3

Overview

3.1 Introduction

The following chapter is an overview of the various facets of this project including the technologies used, techniques implemented, and what licensing this software is released as.

3.2 Backup Strategy

The backup strategy or process that this project implements is as follows. The data to be backed up from users is pulled by actions initiated by the backup daemon. The user shares their data via RSync over SSH and the backup daemon connects to the users' shared data to download it to the

storage pool. Data is stored in a per host ZFS virtual filesystem. Each host backed up receives their own ZFS virtual filesystem. After each new backup set is downloaded, a filesystem level snapshot is taken. The new snapshot is indexed and reported back to the database. Users can then see what files were backed up on a per snapshot basis. Quotas are enforced via ZFS, and compression can also be done by ZFS on the fly. Data is deleted on a configured retention policy. The backup, restore, and deletion tasks are scheduled by a scheduling daemon and executed by the backup daemon.

3.3 Restoration Strategy

Users are presented with a selection of available snapshots to restore. They are then allowed to select files or directories to add to the restore job. File and directories from multiple snapshots are allowed to be selected. Once the needed files are selected, the website creates a restoration job. The backup node reads that a new restoration job has been created, and builds a tar archive of all the files and directories selected. The tar archive is placed in a user accessible location. This location and a base URL is then saved back in the messages section of the restoration job. The web front-end displays the complete URL to the user. The user downloads the tar archive and extracts it on their local computer. At this point the restoration is complete, and the

user has access to needed files and directories.

3.4 Incremental Block-Level Backups with ZFS

Utilizing the snapshot feature in ZFS, a snapshot of the client's files stored in the storage pool is created before each new backup is started. By doing this, a point in time instance of all files currently stored in the filesystem is made. Having a point in time instance allows for restoration of files from any combination of versions of stored files as all versions are available all the time to the backup daemon. Using ZFS snapshots allows for block-level incremental backups. When updated files are copied from the client, only the differences in those updated files will take up additional storage space, thus resulting in less storage requirements.

3.5 Dealing With Backup Failure

Backup failure is not automatically resolved by the backup daemon, but rather reported to the database and subsequently shown to the user or administrator to take appropriate action. In cases where backup jobs are partially completed, the backup is ignored and will be overwritten by the next successful backup job. Subsequent backup jobs will copy additional changes from the client completing partial backups, but will also remove any changes since the last snapshot. No snapshot is taken unless all files are successfully trans-

ferred from the client. Common errors that will cause a backup job to fail partially is permission issues on the client, network issues, empty specified directories (RSYNC assumes this is an error), and authentication errors.

3.6 Load Balancing Between Backup Daemons

During schedule creation, a list of nodes that have the backup service enabled is listed for a users or administrators selection. The user must be aware of the current conditions one each backup node to be able to select the best option. As a result, each node show view includes a graphical representation of the node's used and available disk space as well as current CPU load. Based on this information the user can make a reasonable decision on what backup node to use for his or her new schedule.

3.7 Overall Architecture

In figure 3.1, the general flow of data between entities in the Zackup service is shown. The direction of the shown arrows indicate the direction the data is traveling. This figure visualizes the interaction between entities of the backup service. However, this figure does not show the internal tasks of each entity. Please refer to the previous sections as well as Chapter 5 - Software Entities and Descriptions for more detailed explanation of internal tasks.

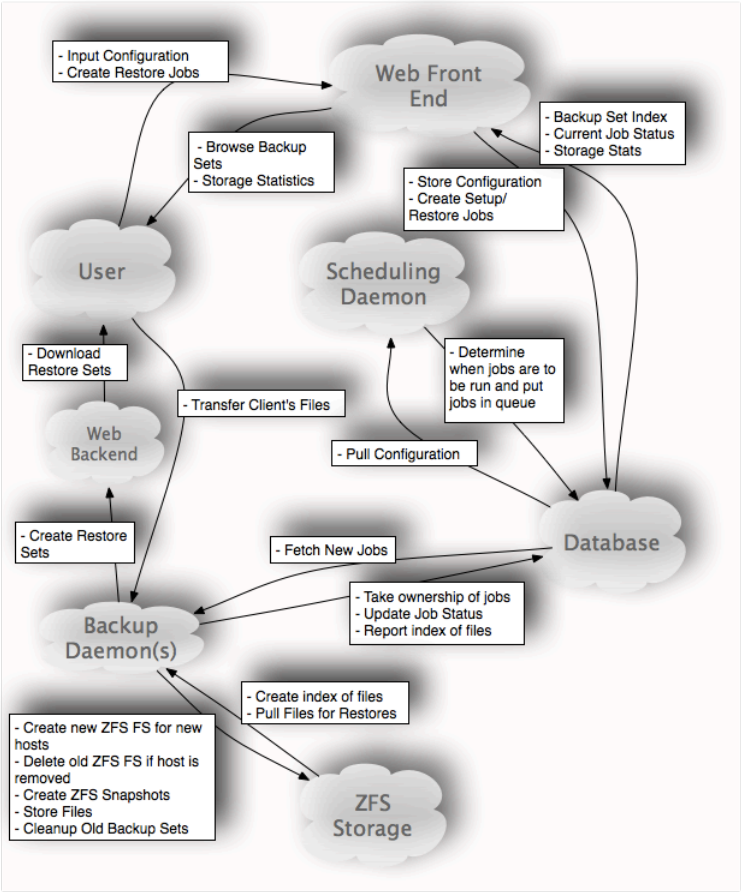

Figure 3.1: Data Flow Between Entities

3.8 Development Language

Ruby 1.9 is the primary language of Zackup. This is for three primary reasons. First, it is the language that the author is most well versed in. Second, Ruby on Rails offers a distinct advantage for producing a web-based front-end, as it offers a solid, well-tested, easy to develop environment to build dynamic websites. Lastly, Ruby is an easy to use and understand language that has many libraries available that simplifies the coding process.

3.9 Operating System

ZFS is a Sun Microsystems' technology and is best supported on Solaris. In addition to Solaris, recently ZFS has been ported to run on FreeBSD, where FreeBSD 8.0 or greater is currently the only operating system outside of Solaris that ZFS is considered stable.

Solaris is available in a few forms. First there is Sun Solaris, second OpenSolaris, and lastly Solaris Express Community Edition. Sun Solaris is Sun's production release of its operating system. Sun Solaris has a very large install base among enterprises, and is widely seen as one of the most stable operating systems available. Its only drawback, as with all operating systems developed to be stable, is that it lags behind in new features.

OpenSolaris is a decently stable operating system that releases a new version about every six months. OpenSolaris' code base is entirely open source, and lacks some of the Sun proprietary programs of Sun Solaris and Solaris Express Community Edition. OpenSolaris offers a mix of new features and stability, but appears to be built primarily as a desktop operating system. The install includes all packages, and no customization during install is available making OpenSolaris a very large installation.

Solaris Express Community Edition (SXCR) is Sun's community development release of its operating system. Currently it is code-named Nevada. A new version of SXCR is released every other Friday. This operating system is a development platform and as such is very dynamic in its support for hardware, new features, and stability. It also contains some Sun software that is not available to the open source community and as such is not included in OpenSolaris.

For its stability, wide user base, and small install size FreeBSD was used for development and testing of the backup and scheduling daemons. The web front-end and database do not necessarily need to be run on a FreeBSD based operating system, but during development they primarily run on FreeBSD for homogeneity of the environment.

Although FreeBSD was the primary development platform for Zackup, it should be fairly portable to other operating systems that have support for ZFS, such as Solaris.

3.10 Hardware

The initial development environment for Zackup was virtual machines (VM) on the NSSA department's RLES system. It required only one virtual machine. This VM had 1GB of RAM, and 80GB of disk space. It served as the web server hosting the web based front-end and hosted the database. In addition, it served the backup daemon and scheduling daemon.

The second development environment for Zackup was a 1U Intel Core 2 Duo server that ran VMware ESXi 4. Again only one virtual machine was used for development. This virtual machine had matched specs with the initial development environment. This hardware was hosted inside the NSSA Labs.

The last and current development environment is in another VMware ESXi environment hosted at the author's current place of employment. It is a two cpu Intel Quad Core CPU machine with 32GB of RAM. Again only one virtual machine was needed.

3.11 Licensing

One of the goals of this project was to be licensed in an open source fashion once the software had been completed, submitted, and accepted for completion of this thesis. This was to be done for few practical reasons. This project takes advantage of many open source programs and as such to license it in a commercial or closed-source fashion would have taken additional effort to verify the legality of doing so. In addition, inevitably the author as the sole developer will face other work commitments, other exciting projects, as well as other time consuming activities. As a result the ability of a sole developer to maintain the software as a matter of course, will wane. Hopefully when released to the open source community, other developers will take an interest in the project and help maintain it. Distributing this work load to the community will be ultimately beneficial to the software. The open source license used will be the MIT License as it matches the Ruby language, and Ruby on Rails web framework. After the acceptance of this thesis, this project will be posted on http://www.github.org/unilogic/zackup for public consumption.

4

Background

4.1 Introduction

A number of topics and technologies used in the design and implementation of Zackup have a long history. The section gives a brief overview of a few main areas.

4.2 Backing Up and Disaster Recovery

As long as computing has been around, and data has been stored there has been a need to backup that data to protect against loss. Backup is a key component of any disaster recovery plan, and simply a necessity in any workplace and home computing environment.

4.3 FreeBSD

"FreeBSD is a freely available, full source 4.4BSD-Lite based release for Intel i386, i486, Pentium, Pentium Pro, Celeron, Pentium II, Pentium III, Pentium 4 (or compatible), Xeon, DEC Alpha and Sun UltraSPARC based computer systems" [3]. It initially grew out of the "Unofficial 386BSD Patchkit", with an initial 1.0 release in December of 1993. Currently FreeBSD's release is at version 8.0, with a legacy release of 7.2 still maintained. FreeBSD is a widely used and considered extremely stable and lightweight.

4.4 ZFS

ZFS was originally announced by Sun Microsystems, Inc. on September 14, 2004 and integrated into Solaris development on October 31, 2005 [2]. "Originally, ZFS was an acronym for "Zettabyte File System." The largest SI prefix we liked was 'zetta'. Since ZFS is a 128-bit file system, the name was a reference to the fact that ZFS can store 256 quadrillion zetabytes" [8]. ZFS has become arguably the most dynamic and useful filesystem available.

4.5 Ruby and Ruby on Rails

Ruby is a relatively new language, initially started in 1995 and widely adopted in 2006. There are many available libraries for Ruby simplifying programming. With an extremely active open source community it is con-

tinually improving. Ruby on Rails, a widely adopted web framework, it is also a very active open source project with many updates and improvements continuously being made. Overall, Ruby and Ruby on Rails offers one of the best language and frameworks in aspects of usability, ease of programing, and forecasted continuation of adoption and support. It is also released under the MIT license, which allows liberal reuse of the project, and little restriction in commercial or open source uses of the code.

4.6 Summary

Backup, FreeBSD, ZFS, Ruby, and Ruby on Rails are all integral parts of the Zackup service. Without any one of these parts, Zackup's design would have taken a different route.

5

Software Entities and Descriptions

5.1 Introduction

The descriptions of software entities that were developed for Zackup are detailed in the following section. Related designed decisions, including justification for those decisions is included.

5.2 Database

The backup service is database agnostic and supports almost any database. It uses Ruby on Rail's ActiveRecord library that is a abstraction layer for various popular open source databases like MySQL, PostgreSQL, and SQLite3. The database is used to store all configuration information for the backup ser-

vice, including the web front-end, backup daemon, and scheduling daemon, as well as status information of waiting, running, and finished jobs, statistics on storage pools, and lastly backup set file indexes.

5.3 Web Front End

The primary purpose of the web front end is to allow configuration of the service. To allow for restoring of files a file browser is included as seen in figure 5.1. It allows the browsing of all files of all backup sets for each schedule defined for a host. The user is able to select individual files or folders from any backup sets available to create a restore job. The web front-end provides feedback to the user and administrator in the form of node specific storage pool usage statistics and per schedule quota usage statistics. Lastly, user administration is completed from the web front-end, allowing users to be registered, added and removed.

Figure 5.1: Restore File Browser

5.3.1 Configuration Items

A configuration item is a defined attribute used to add a needed or useful bit of information for a host. The configuration items are defined in a hierarchical manner. Configuration items are broken into three categories. First default configuration items, these items are common to all hosts, and are automatically presented to a user to configure when creating a new host. These include IP address, hostname, backup directories, etc. These items are all required no matter what type of client is to be created.

The next type of configuration item is a host type item. There can be multiple host types. Host type are based on the file transfer method used for the client. For example when using RSYNC over SSH the SSH host type is selected. Under the host type is any specific configuration items to that service. In the case of SSH, ssh_login, ssh_port, ssh_private_key, and ssh_private_key_password are all included.

The third type of configuration item is an option. These are intended to be more generic configuration items that are not necessarily needed for each host, and do not related to the type of file transport to be used. This feature is implemented, but no examples have been identified for practical use.

Additionally to having three type of configuration item types, a number of configuration items are hidden, or in Zackup parlance not configurable. These items are not available to the user when the host is being created or edited, and is used by the service to store information about the host. In addition, these hidden items can also be used as the parent for groups of items used for host type items.

5.3.2 Configuration Item Tradeoff

Configuration items as described in the previous paragraphs allows for an extremely dynamic web user interface. As new file transport mechanisms

are implemented or other configurable item is needed, an administrator can quickly add that feature to the website without editing any code. This specific reason was the rational for its implementation. As the web front-end was the first entity of the Zackup service to be implemented, the configurable inputs needed for the various daemons was unknown.

To achieve the configuration item design, the Ruby and Rails framework had to be pushed outside of its norm to work with a fully normalized database that included a weak entity table called host_configs. This table stored an actual value of a configuration item for a host. It as well related the config_items table, which stores the name, description, HTML form input type (i.e. password field, text area, text field, etc), and whether the item was configurable to the hosts table which stores the host's friendly name and an id. As a result of using three tables to achieve dynamic configuration items, whenever a value of a host's specific configuration item wanted to be retrieved, an expensive SQL JOIN query must be made. Moreover this query must be repeated for every configuration item retrieved. As a result, the host's index view, which shows all the hosts with their IP address, hostname, and status is extremely inefficient in its database use. This inefficiency is the tradeoff for having highly dynamic and customizable interfaces. Since performance was

not a specific goal of this project this inefficiency was tolerated. Ideally all of the config_item table would be cached in memory as it is rarely modified. The host_config table would need to be analyzed for its churn rate (amount of modification) to identify the effectiveness of caching.

5.4 Scheduler

The scheduling daemon is primarily responsible for reading configuration information from the database, and creating new jobs for the backup daemon. The scheduling daemon takes advantage of the Rooster Daemon library. This library is a framework for Ruby based daemons, that allowed TCP based control of the daemon. The addition of a control method allows the web front-end to restart the daemon if needed.

5.4.1 Parsing of Schedules

On each iteration of the scheduling daemon, a number of checks are performed before new jobs are created from schedules. First a check for how many erred jobs for a given schedule is run. If it is above the configured amount, no new jobs are created. Next, if a job is currently in any state other than erred or canceled it is considered in a running state. If any job is in the running state for a schedule no new job will be created unless a configured option to force job runs is enabled. Next, a check for new backup

storage directories reported by the backup daemon is done. If for some reason a schedule has a new backup storage directory the Scheduling Daemon skips the schedule and reports the error, as backing up to multiple places is a unhanded exception. Lastly, a check is run to ensure a schedule has a retention policy in place.

After the checks are successfully run, a schedule is parsed to determine if based on its last finished time or last started time a new job needs to be created. Last finished or last started time is used depending on a service wide configuration option. The interval for a schedule is calculated into the number of seconds of the interval and compared against the delta of either last started or last finished and the current system time. If the delta is greater then the computed interval, i.e. enough time has passed that a new backup is needed, a new job is created. If a schedule does not have a last started time or last finished time, in other words its a new schedule, the job will be run at the schedule's specified start at time. A schedule can postpone the first backup job to a certain time and date by specify the start at time in the future. One last check is done before the job is saved to the database. If the calculated start time of the new job is not within one iteration time period of the scheduling daemon from the system's current time the job will not be

saved. This is to eliminate crowding of assigned jobs in the database. If the job's start time is within one iteration time period the job or is in the past is saved to the database for the backup daemon's consumption.

5.4.2 Parsing of Finished Jobs

Jobs marked finished by the backup daemon, are parsed for any interesting information by the scheduling daemon. Information items includes new backup storage directories from setup jobs, which is the physical paths of the ZFS filesystem on the backup daemon used for the schedule. In addition, a schedule's last finish time is updated for each newly finished job. All other data from the backup daemon, such as file indexes, is directly stored in other tables of the database by the backup daemon itself.

5.4.3 Startup configuration

Due to some limitations in the daemon library used for the scheduling daemon and the Rails libraries. Any configuration items there were to be read on startup of the daemon could not be read from the database. As a result a YAML file, a tab delimitated method of storing hash tables as a string, was used.

An example of this configuration file that has separate values for different running environments looks like figure 5.2.

```
development:
  parse_interval: 60

test:
  parse_interval: 60

production:
  parse_interval: 60
```

Figure 5.2: Example Scheduler YAML Configuration File

5.4.4 Independent vs Integrated With Website Code Base

A decision was made early in the design of the scheduling daemon whether to tightly integrate it with the source code of the web front-end or to develop it completely independent. To reduce redundant code the scheduler daemon was integrated within the web front end code base. It reuses much of the Ruby on Rails framework from the web front end for database connections. No use case was identified where multiple scheduling daemons independent from a website would be needed. While untested at this stage, a second website and scheduling daemon using the same database as the first should be possible. Additional testing to see if multiple websites and scheduling daemons interacting with the same database would require locking of in-use data is needed.

The only downside of using Ruby on Rails code for the scheduling daemon is the loading of unneeded libraries inherently included with Rails and its environment. This resulted in a larger than anticipated memory footprint of the relatively simple scheduling daemon.

5.5 Daemon for Backup Tasks

The backup daemon is primarily responsible for interacting with the ZFS storage pool's filesystem and transferring clients files to that filesystem. It ended up being the most programmatically complex piece of software created in this project. The backup daemon's tasks include:

- Downloading files from users
- Interacting with ZFS
  - Creating Snapshots
  - Creating ZFS Filesystems
  - Setting ZFS Attributes
  - Cleaning Up Old Snapshots Based on Retention Policy
  - Indexing Files in a Snapshot
- Polling the Database for New Jobs
- Updating the Database with Job Status
- Updating the Database with Backup Set File Index
- Updating the Database with Error Status of Jobs
  - RSYNC and SSH Errors
  - Other System or Connection Errors

- Creating Restore Sets and Making Them Available via the Web Back End

5.5.1 Backup Jobs

The Backup Daemon used standard object oriented programming. Described in the next few paragraphs is how the backup daemon logically builds and executes a backup from a backup job.

First a new BackupJob object is instantiated with the following attributes: hostname, host_type, local_backup_dir, exclusions, directories, and port number. These attributes are garnered from the job's data column, which is a YAML'ized, i.e. stored in YAML format, hash. From the BackupJob object an Rbsync object can be created. Each of these objects are a different level of abstraction. The backup job requires the above attributes, while the Rbsync object actually builds the command line string that will be called to run the Rsync binary.

In addition, the Rbsync object decrypts and writes out the client's SSH key to a temporary file. This is done for SSH's benefit. As the Rbsync module uses SSH for Rsync's transport method, a method of authentication was needed. As passwords cannot be easily passed to the SSH binary, and a path to a SSH private key can, SSH keys were used. For the key to be used

though, it must be written out to a temporary file in an unencrypted fashion. As a precaution, the temporary key file is ensured to be cleaned up after use to limit its exposure.

After the Rbsync object creates a command line string from the given attributes, an instance method named pull, executes the command. It also captures the output of the command as well as the return status of the command. Both of these are used to determine if the command failed or succeeded.

After the Rbsync runs the Rsync binary, it checks the commands return status. If the command returns zero for success, a ZFS snapshot is created for the filesystem. The snapshot is named the current system time. As long as the snapshot succeeded, a file index of the newly created snapshot is created. This is done by traversing the snapshots directory in the ZFS filesystem. The snapshots directory exists inside the .zfs directory at the root of the filesystem. The file index is stored as a hash data type, where directories are represented by a key that has a non-null value, and a file is represented in by a key that has a null value. A custom use of the hash datatype, as seen in figure 5.3, was used that allowed nested hashes to be created on the fly without creating the parent element. In other words. a hash could be specified like

`file_index['usr']['local']['etc'] = { httpd.conf => nil }` without first creating `file_index['usr'] = {'local' => {}}` and so on. The former being much easier to implement based on file paths and string manipulation. Figure 5.3 is the full implementation of the class that is responsible for indexing snapshots.

```
require 'find'
require 'zlib'
require 'yaml'

class CustomFind
  def self.find(path, basepath=Dir.pwd)
    Dir.chdir(basepath) do
      data = Hash.new { |l, k| l[k] = Hash.new(&l.default_proc) }
      Find.find(path) do |path|
        path_split = path.split('/')
        dest = ""
        path_split.each do |base|
          dest << '["' + base + '"]'
        end
        if File.directory?(path)
          eval("data#{dest} = {}")
        else
          eval("data#{dest} = nil")
        end
      end
      zdirs = Zlib::Deflate.deflate(YAML::dump(data), 9)
      return zdirs
    end
  end
end
```

Figure 5.3: CustomFind Class

After the file index is created, it is compressed with the Zlib library, and then stored in the FileIndex table as a binary datatype for the web front-end's consumption. The job is now marked as finished if all previous ran commands returned correctly, otherwise the job is marked as erred with the relevant error message and return status.

5.5.2 Restore Jobs

A restore job is created by a user from the web interface by selecting various files and directories and submitting the job to the database. When the Backup Daemon finds a restore job in the assigned state, it creates a Tar archive of the given directories and files. Finally it makes the Tar archive available for consumption by the user via the web back end.

Logically the restore process goes as follows. A RestoreJob object is created with a hash that describes the directories and files and their belonging snapshot. The RestoreJob then iterates through all the directories and files, placing them in a newly created Tar archive. The Tar archive is compressed using the Zlib library, and is moved to a configured storage location on the server. This location, as well as the base of a user accessible URL is configured on the Backup Daemon during startup. The Backup Daemon now saves the full URL for the restore set back to the database in the data section of

the appropriate job. It marks the job finished if all commands ran correctly. Otherwise, it will mark the job erred with the relevant error message and return status.

5.5.3 Maintenance Jobs

A maintenance job's function is to destroy expired ZFS snapshots. ZFS snapshots are to be determined expired by the scheduling daemon based on a schedule's retention policy.

The Backup Daemon creates a MaintenanceJob object with the snapshots to be deleted. It then calls the appropriate ZFS commands to destroy those snapshots. The job is marked finished if the commands successfully ran. Otherwise, the job is marked erred with the relevant error message and return status.

5.5.4 Setup Job

A setup job is responsible for creating the ZFS filesystem for a new schedule. It also takes care of updating any needed settings for the new ZFS filesystem.

A SetupJob object is instantiated with IP address, hostname and quota attributes. As long as the attributes are supplied the new ZFS filesystem is created. The pathname to the new filesystem is saved back to the relevant job for the Scheduling Daemon's consumption. The job is marked finished if

all commands execute successfully. Otherwise, the job is marked erred with the relevant error message and return status.

5.5.5 Threaded vs Non-threaded

One key design decision for both the backup daemon as well as the scheduling daemon was if they were to be designed in a threaded or multi-process fashion versus a serialized designed. A serialized design being where only one action happens at a given time, all actions happen in a row and not in parallel.

There is a few classic tradeoffs between the two designs. A serial design is simpler, easier to debug, and reduces the chances of unforeseen errors. However, it allows only one task to occur at a time. As a result jobs can become back logged under heavy workload.

Conversely, a threaded design would allow multiple jobs to be run in parallel, thus increasing the ability of the Backup Daemon to do work. This has the potential to use more systems resources and must be carefully monitored. Also monitoring thread status and ensuring threads exited correctly is more complex than the serial design. This topic is more thoroughly discussed in the Future Work chapter, section 9.3.2.

5.5.6 ZFS Ruby Library

Initially a C based Ruby extension library named Zetta, available here: http://github.com/rubaidh/zetta, was considered for interfacing with ZFS directly. However the project has not been updated since June of 2007, and does not compile on FreeBSD. In addition, it appears to be lacking some key features needed from ZFS like ZFS's snapshots.

As a result of not suitable ZFS libraries being identified, it was determined the best course of action was to write a Ruby wrapper library around the ZFS command line utilities. This decision was made for a few reasons. First the author knows Ruby and not C, so attempting to update the existing library would have been more time consuming. Second, FreeBSD's implementation of ZFS seems to be significantly different from Solaris. Thus compiling a Ruby library that depends on ZFS related libraries and headers on both systems would be significantly difficult as well is hard to maintain. Lastly, Sun has noted the output of the command line utilities are to remain stable allowing reasonable assurance that wrapper libraries that parse the text of the output from the utilities will not break with future releases of ZFS.

It turned out that writing the Ruby ZFS wrapper library was fairly trivial. The command line utilities outputted text in a way that was easy to manip-

ulate into useful data structures. Column headers were used as keys for hash tables. Each row of the output was a multi-key anonymous hash nested in an array. This implementation proved to be completely reliable through out the development of this project.

5.6 Web Back End

The purpose of the web back-end is to serve restore sets to the user after they have been created by the Backup Daemon. The web back-end is typically run on the same server as the backup daemon as the Backup Daemon needs filesystem access to the published area of the web back-end. The web back-end is typically a light web server that is efficient at serving files from the filesystem. The Nginx (pronounced engine x) is suited well for this task and is available from http://wiki.nginx.org/Main. When all entities of the Zackup service are run on a single server, the web front-end and web back-end can use the same web server without issue.

5.7 Inter-entity Communication

The scheduling daemon and backup daemon communicate via a job queueing mechanism where the database queue or store jobs produced by the scheduling daemon or web front-end in the Jobs table. The backup daemon periodically will poll the database for currently available jobs. If the

Attribute	Data Type	Summary
backup_node_id	integer	Corresponds to the ID of the backup node to perform the work.
scheduler_node_id	integer	Unused, in the future possibly used for multiple scheduling daemons.
status	string	Status of the job.
created_at	datetime	Date and Time the job was created.
updated_at	datetime	Date and Time the job was last updated.
data	text	Serilized hash of various configuration items, and returned data from the Backup Daemon.
host_id	integer	the ID of the host this job is related to.
schedule_id	integer	the ID of the schedule this job is related to.
finished_at	datetime	The date and time the job was marked as finished.
start_at	datetime	The date and time the job is to be started at.
operation	string	What type of job this is: setup, backup, restore, or maintenance.

Figure 5.4: Job Table Structure

specific backup daemon can process the job, denoted by matching node IDs in the job and the configured node for the Backup Daemon, it marks the job as running. The daemon then processes the job. The backup daemon updates its progress on jobs as it finishes with them. The job table's structure is shown in figure 5.4.

5.8 Summary

The Software Entities and Description chapter discussed each individual part that makes up the Zackup service. The main entities Web front-end, Scheduling Daemon, Backup Daemon, and Web back-end were broken down into their individual tasks, and how those tasks were performed. Tasks were described how they were logically accomplished in code. Creative and other interesting tidbits of code were showcased for their use.

6

Findings

6.1 Introduction

Part of the Zackup project was to identify how ZFS can be used to enforce a backup service's retention policy. In addition, Zackup once completed was put through a simple test to ensure its stability. Both of these studies are detailed in this chapter. Zackup will be shown to be a stable and usable service.

6.2 Applicable Retention Policies Allowable Using ZFS Snapshots

The following section describes what retention policies or usable methods to effectively expire unneeded backup data, as well as how to maintain a minimum amount of backup data to ensure proper backup coverage. One thing that should be mentioned before going on is that backup data can only be expired on a per snapshot basis. In other words, all the below retention policy methods are based on calculating the time or version number of a snapshot as a whole. Individual files cannot be effectively expired using the ZFS filesystem's built-in utilities.

6.2.1 Keep Data For At Least N Time Units

Keep data for at least N time units is a retention policy method where an individual snapshot will be protected from expiration for a set amount of time. This method is possible by recording the date and time of snapshot creation, and investigating that time versus the current system time. When the difference between the two times is greater than the configured N time units, the snapshot is no longer protected from expiration. If no other retention policy method is set for the specific snapshot, it will be expired immediately after it is no longer protected. This method sets a minimum age

for a snapshot.

6.2.2 Keep Data No More Than N Time Units

Keep data no more than N time units is a retention policy method where an individual snapshot will be expired after the configured N Time units has passed from the snapshots creation date and time. This method sets a maximum age for a snapshot. Similar to the above method, creation time is recorded and upon investigation is compared against the current system time. When the difference is greater than the configured N time units, the snapshot is expired. However, if the snapshot to be expired is protected by either the Keep data for at least N Time units method or the below Keep at least N versions method it will not be expired until that protection has been removed. If no other method is configured, this retention policy method will expire snapshots as described.

6.2.3 Keep at Least N Versions

Keep at least N versions, is a simpler retention policy where the number of snapshots are counted. If there is at least the number configured versions of snapshots that are newer than the investigated snapshot, that snapshot will no longer be protected from expiration. This method sets the minimum number of versions of snapshots for the backup. If no other method is config-

ured, this retention policy method will expire snapshots as soon as they are no longer protected.

6.2.4 At Max Keep N Versions

At max keep N versions is similar to the previous Keep at least N versions method, where the number of newer snapshots than the investigated snapshot is counted. However, if the number of newer versions is greater or equal to the configured number of versions, the snapshot will be expired. Keep data for at least N time units and Keep at least N versions will override any snapshots expired by this method. If this retention policy method is configured alone, it will expire snapshots as described.

6.2.5 Implemented Retention Policy Methods

The above four retention policies were implemented in Zackup, and function as described. Time units can be specified in hours, days, weeks, months, and years.

6.3 Testing

6.3.1 Description of Testing

The test involved running the Zackup service with two clients over a one week period. One client was a Mac OS X workstation actively used by the author,

and the second client an active development source code management server used by the author and his co-workers. The intention of this test was to ensure the daemons will run for a longer period of time without depleting server resources or running into unforeseen fatal errors. In addition, any errors that occurred were investigated and noted in the reliability section 6.3.2. Job completion time of the four job types were recorded and analyzed for general patterns. To study the performance of the Zackup service cpu, memory, and disk consumption were looked at throughout the week of study. Lastly, to see if the goal of "contain[ing] substantially less author created code than a comparable service that implemented backup sets by hand" was met, the number of lines of created code will be counted and compared to the similar project BackupPC.

6.3.2 Reliability

An error was found during this test that resulted in the Backup Daemon failing fatally a random amount of time after start up. The Backup Daemon initially used the Sys/CPU library to retrieve current CPU load averages, which either has a conflict with Ruby 1.9 or FreeBSD 8.0, and caused a fetal error. The problem was corrected by simply parsing the uptime command line utility for the current CPU load averages.

An additional typical error made its presence with the setup of the server client. A number of backup directories specified had subfolders that had different permissions, and as a result Rsync returned a error 23, partially downloaded return status, on the first backup job. This error was easily corrected, by changing the permissions on the directories were possible and adding exclusions to the host.

Other than the noted issues during setup, the service ran flawlessly through the entire test.

6.3.3 Job Completion Time

Total of 252 backup jobs ran. The average time of completion was 35.22 seconds. The longest job took 59.22 seconds, and the shortest job took 19.52 seconds. Job completion time was calculated based on time the job was created and the time the job was marked finished by the Backup Daemon. The gathered data is shown in figure 6.1 with a linear line of best fit running right around the average time.

The service was backing up two clients with datasets of about 2.5GB as seen in the following performance section. It appears that data churn was relatively low on these clients, so backup jobs took on regular intervals. Overall the backup times seen here were very acceptable.

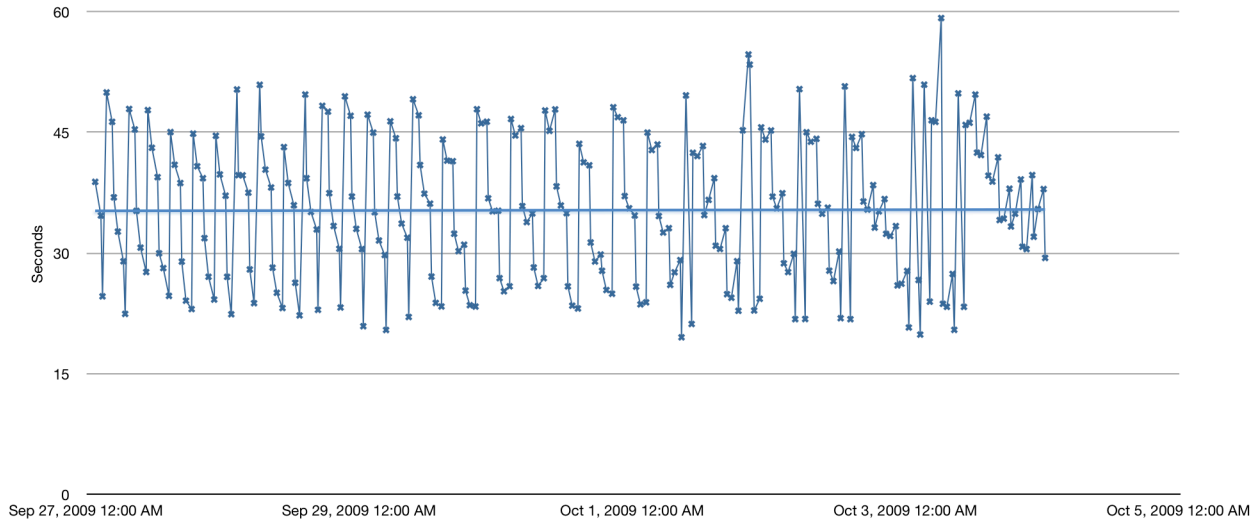

Figure 6.1: Job Completion Time Graph

6.3.4 Performance

The current CPU usage was recorded with each iteration of the Backup Daemon. This data is shown in figure 6.2. This shows relatively little utilization of the CPU with a few peaks around 3AM on most nights. This 3AM time coincides to when FreeBSD runs a set of daily maintenance tasks by default, and is likely attributable to this. Overall CPU usage seen is very acceptable often showing just above idle. Zackup has very little affect on CPU resources on a machine. On average the CPU load seen was 0.026, with a maximum of 1.32 and a minimum of 0. Measurements were taken from the 1 minute load average shown by the uptime utility about every minute.

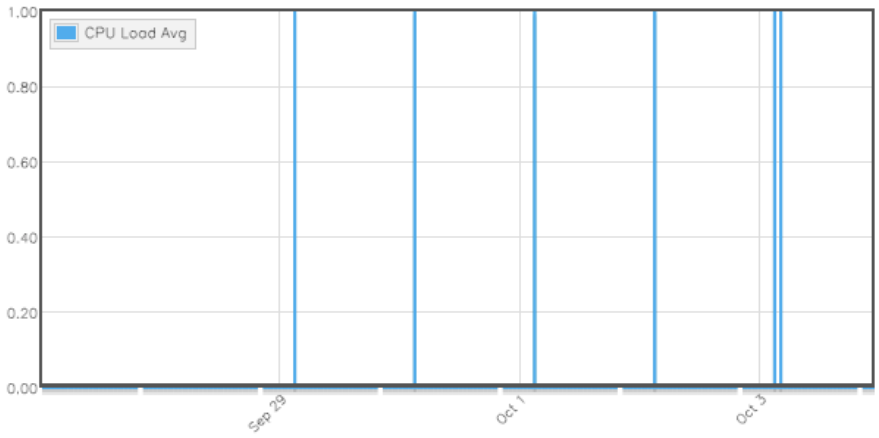

Figure 6.2: CPU Load Usage Graph

Each entity in the backup service took a certain amount of memory space. At the end of the testing period each entity was checked for its current usage. This usage should be whats expected with Zackup running but not currently running any jobs. As jobs run memory usage will fluctuate. The Scheduling Daemon used 182MB of RAM. The Backup Daemon used 145MB of RAM. The web front-end used 118MB of RAM. A grand total of 445MB of RAM. This number seems a bit high, but is likely accurate. Ruby is not the most efficient language to use when lowest memory use is desired. Moreover, all entities listed are using at least parts of the Rails framework. The Backup Daemon is just using ActiveRecord. The inclusion of this relatively large framework explains a lot of the memory bloat seen.

The average disk spaced used for backups was 5.195 GB. With a minimum of 2.492 GB and a max of 5.276 GB. From these stats alone it is shown there was very little data to backup after the initial backup was completed. To further illustrate this figure 6.3 shows disk available and disk usage over time. The only noticeable change at that scale is during the first backup of data from the clients.

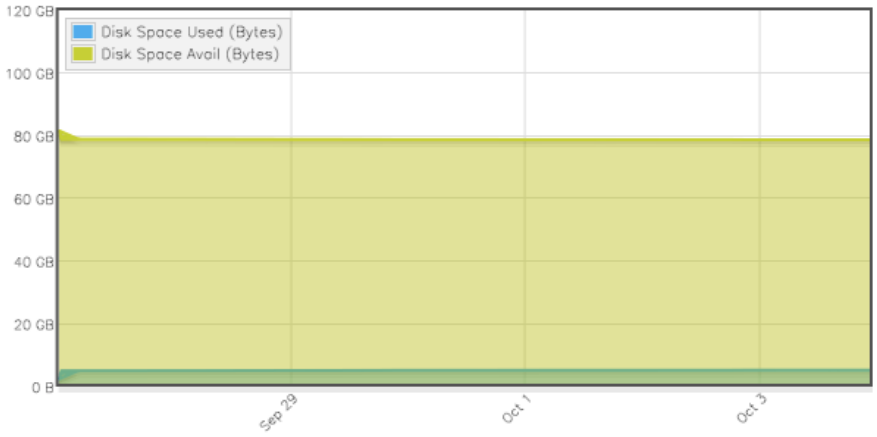

Figure 6.3: Disk Usage Graph

In figure 6.4 the output of `zfs list` is shown. This displays the current used and available disk space. Note the available disk space differs from the two clients as each had a differing quota value.

6.3.5 Size of Program

Comparing number of lines of code is not considered the most accurate way of determine the size of programs, but offers a rudimentary method

```
NAME                          USED   AVAIL  REFER  MOUNTPOINT
backup                        4.91G  73.3G    22K  /backup
backup/192.168.1.146_BtAnHl   2.59G  7.41G  2.43G  /backup/192.168.1.146_BtAnHl
backup/192.168.1.244_FerYRY   2.32G  27.7G  2.32G  /backup/192.168.1.244_FerYRY
```

Figure 6.4: Zfs List Showing Disk Usage

of measure. In addition, BackupPC is a much more mature program than Zackup. As such is has localizations to a varying number of languages, full documentation, and has a number of features Zackup does not. For these tests both the localizations (./lib/BackupPC/Lang) and documentation (./doc) were left out of measurement. Remember these tests are very rough, and should be used for generalizations not exact data points.

```
find . -type f -not -path "./.git/*" -not -path "./vendor/*" -not -path "./test/*"\
-not -path "./script/*" -not -path "./public/javascripts/*" -not -path "./log/*"  \
-not -path "./public/*" -not -path "./db/*.sqlite3*" -print0 | xargs -0 wc -l
```

Figure 6.5: Find Command to Show Author Created Code in Zackup

Zackup's total author created lines was **9120 lines**. Note CSS files were not included in find command seen in 6.5 and were manually added to the total.

BackupPC 3.2.0 Beta was downloaded from http://iweb.dl.sourceforge.net/project/backuppc/backuppc-beta/3.2.0beta0/BackupPC-3.2.0beta0.tar.gz. With the aforementioned directories excluded a total of 32356 lines

were counted.

Assuming all code in BackupPC was created by the project's authors and not 3rd party libraries, and given a huge margin of error of 50%, Zackup is just over half the size of author created code as BackupPC.

6.4 Summary

ZFS provided an ample way of enforcing retention policies with four distinct methods. Zackup proved to be a reliable service during a one week test with two clients. While the clients used did not provide a heavy load to the Zackup service, it did show that Zackup was capable of running unassisted for a longer length of time. Since reliability is a key factor in any backup service, Zackup's performance in this respect is indeed important. Zackup's resource consumption is a bit mixed. It used very little CPU resources, but uses an unexpected amount of memory. While the memory usage is not unacceptable, future versions of Zackup will work on reducing this number. Disk consumption was as expected, with little change after initial backups. Lastly the size of Zackup compared against BackupPC was satisfactory. Overall, Zackup meets the goal of "contain[ing] substantially less author created code than a comparable service that implemented backup sets by hand", and appears to be a stable backup platform.

7

Install Documentation

7.1 Introduction

The install documentation chapter runs the administrator of the Zackup service through the install and setup process of the service. The documentation starts with a base FreeBSD system, and takes the administrator all the way through setting up a client for backup.

7.2 FreeBSD Base Install to Ready for Zackup

At this point this guide assumes there is a working FreeBSD 8.0 install with the ports tree already installed. This guide will use the entirety of a second hard drive in the server. Please adapt the following ZFS Pool Setup section

to the available storage devices.

7.2.1 ZFS Pool Setup

1. First ensure ZFS is loaded into the kernel by running the following two commands

   ```
   # echo "enable_zfs=yes" >> /etc/rc.conf
   # /etc/rc.d/zfs start
   ```

2. The next step is to create the zpool. For the second hard drive device **/dev/da1** and the zpool name backup run the following

   ```
   # zpool create backup /dev/da1
   ```

3. Run `zpool list` to ensure the new zpool was successfully created.

   ```
   $ zpool list
   NAME     SIZE   USED  AVAIL    CAP  HEALTH  ALTROOT
   backup  79.5G   18k   79.5G     0%  ONLINE  -
   ```

4. Delegate permissions, the following commands delegate all the necessary permissions to the user **zackup** for the Zackup service to run a non-root user.

   ```
   # zfs allow zackup create backup
   # zfs allow zackup destroy backup
   # zfs allow zackup mount backup
   # zfs allow zackup snapshot backup
   # zfs allow zackup canmount backup
   # zfs allow zackup mountpoint backup
   # zfs allow zackup quota backup
   ```

```
# zfs allow zackup snapdir backup
# sysctl -w vfs.usermount=1
```

7.2.2 Preparation For Installing Needed Ports

In preparation to install the needed programs to support Zackup the following command needs to be run.

```
# echo "RUBY_DEFAULT_VER=1.9" >> /etc/make.conf
```

7.2.3 FreeBSD Ports

FreeBSD's ports tree offers a way to easily compile and install a number of third party programs thats not included in the core of the operating system. To install a port, change directory into the corresponding directory, ex. `cd /usr/ports/devel/git`. Then run `make && make install`. This will install and compile the port.

Install the following ports.

`devel/git`

`lang/ruby19`

`convters/ruby-iconv`

`database/postgresql` (Mysql or any other ActiveRecord supported database will suffice, SQLite won't work well if both daemons are accessing it at the same time.)

`net/rsync`

7.2.4 Ruby Gems

Ruby third party libraries are managed by the **gem** command. It is used to install, update, and uninstall Ruby libraries. Before you begin, a number of the libraries are hosted on Github and as such Github must be added as a source for gems by running `gem sources -a http://gems.github.com`. Gems are installed by running `gem install <gem_name>`. The following gems are needed to run the web front-end and the scheduling daemon.

- `rufus-scheduler`
- `aasm` (**Note:** The author had to download the source and compile this gem by hand.)
- `authlogic`
- `calendar_date_select`
- `mbleigh-seed-fu`
- `ezcrypto`
- `mislav-will_paginate`
- `chronic`
- `rails`
- `ruby-pg` (**Note:** Pick whichever ActiveRecord adapter needed for your database.)

Hint: In FreeBSD to get the ruby-pg gem to install, run

```
# gem install ruby-pg -- --with-opt-dir=/usr/local
```

7.2.5 Ruby Gems for Backup Daemon

The following gems are needed specifically for the Backup Daemon to be run.

- `daemon-kit`
- `aasm` (**Note:** The author had to download the source and compile this gem by hand.)
- `activerecord`
- `archive-tar`
- `archive-tar-minitar`
- `sys-cpu`

The FreeBSD system should now be ready for Zackup.

7.3 Installing Zackup

Now that the system is ready for Zackup, this section will show how to install the Zackup service, and get all of its entities up and running. The following example will assume the most simple configuration, where all services run on a single server.

7.3.1 Obtaining the Source

The Git distributed source management system is needed to fetch Zackup's source code. It is also possible to fetch a archived version from Github directly

at http://github.org/unilogic/zackup.

To fetch the source run the following commands.

```
# git clone git://github.org/unilogic/zackup.git
# git submodule init
# git submodule update
```

Zackup's source should now be copied to the current directory in a folder named **zackup**.

7.3.2 Website Initial Configuration

Zackup has a number of configuration files that need to be setup. All path names are relative to the root the project.

The **config/database.yml** configuration file is used to configure how the web front-end and Scheduler Daemon access the database. Figure 7.1 shows an example database.yml file. Fill out the attributes for each environment that will be used.

```
development:
  adapter: postgresql
  encoding: utf8
  database: zackup_development
  username: pgsql
  password:

production:
  adapter: postgresql
  encoding: utf8
  database: zackup_production
  username: pgsql
  password:
```

Figure 7.1: Example Database.yml Configuration File

There is a number of pieces of data that need to be pre-populated into the database. This data deals mainly with configuration items. Ensure your database has the database created before proceeding, ex. in PostgreSQL `CREATE DATABASE zackup_production`. Note, SQLite3 database files will be created automatically, but the use of SQLite3 is not recommended for anything but testing.

To create all the needed tables in the database run from the application root:

```
#rake db:migrate
```

Next to insert the pre-populated data run from the application root:

```
#rake db:seed
```

The web front-end and scheduling daemon are now ready to be started.

7.3.3 Backup Daemon Configuration

The Backup Daemon has two configuration files that need to be edited. The **config/database.yml** and **config/settings.yml** files that are located in the backup_daemon subfolder of the project. Note that the backup_daemon directory wholly contains the Backup Daemon and can be moved separate from the rest of the code base. Figure 7.2 shown an example config/settings.yml file. Each item needs to be configured to match the installation environment.

Note: the hostname must resolve to the IP address given or the Backup Daemon will refuse to run. The system's **/etc/hosts** file can be used to archive this if the server's hostname does not resolve via DNS.

```
development:
  # Settings that match the node configured.
  ip_address: 127.0.0.1
  hostname: localhost

  # How long the daemon sleeps after each run
  loop_interval: 30

  # Zvol Used to Store all Backups
  backup_zvol: backup

  # Match this timezone to the one selected in Rail's environment.rb
  time_zone: 'Eastern Time (US & Canada)'

  # For Restores, ensure the trailing slash is included!
  download_dir_base: '/usr/local/www/zackup/public/restores/'
  download_url_base: 'http://hostname:3000/restores/'

production:
  # Settings that match the node configured.
  ip_address: 127.0.0.1
  hostname: localhost

  # How long the daemon sleeps after each run
  loop_interval: 30

  # Zvol Used to Store all Backups
  backup_zvol: backup

  # Match this timezone to the one selected in Rail's environment.rb
  time_zone: 'Eastern Time (US & Canada)'

  # For Restores, ensure the trailing slash is included!
  download_dir_base: '/usr/local/www/zackup/public/restores/'
  download_url_base: 'http://hostname:3000/restores/'
```

Figure 7.2: Backup Daemon Example Settings.yml

The second configuration file is **config/database.yml**. This file specifies how the Backup Daemon reaches and authenticates with the database. More than likely this file should be identical to the web front-end's database.yml file in figure 7.1. **Note:** If using SQLite3 ensure the path to the database file in this configure file is full qualified.

```
development:
  adapter: postgresql
  encoding: utf8
  database: zackup_development
  username: pgsql
  password:

production:
  adapter: postgresql
  encoding: utf8
  database: zackup_production
  username: pgsql
  password:
```

Figure 7.3: Backup Daemon Example Database.yml

At this point the Backup Daemon is configured.

7.3.4 Run Backup Daemon

Note: Before starting the Backup Daemon additional configuration is necessary via the web front-end. Please continue on to the next sections.

1. To start the Backup Daemon, first change directories into the **bacup_daemon** folder located in the base of the project.

```
[/usr/local/www/zackup/backup_daemon]\$ ./bin/backup_daemon
initializer.rb:178: DaemonKit (0.1.7.10) booting in development mode
initializer.rb:357: Setting up trap for USR1
initializer.rb:357: Setting up trap for USR2
initializer.rb:357: Setting up trap for HUP
initializer.rb:357: Setting up trap for INT
initializer.rb:357: Setting up trap for TERM
database_setup.rb:2: Setting up ActiveRecord Connection
initializer.rb:113: DaemonKit (0.1.7.10) booted, now running backup_daemon
backup_daemon-daemon.rb:37: Startup Checks Running
backup_daemon-daemon.rb:49: I am node id: 1
backup_daemon-daemon.rb:79: I'm running
```

Figure 7.4: Backup Daemon Starting Up

2. Then run, `./bin/backup_daemon`. The daemon should start as seen in figure 7.4.

3. The previous command will run the daemon in the foreground, which is good for debugging and ensuring its running correctly.

4. To run the Backup Daemon in the background run, `./bin/backup_daemon start`.

7.4 Configuration from the Web Front-End

A number of first steps much be undertaken using the web front-end to allow the Backup Daemon to successfully be run. Proceed to the following steps to start the web front-end and finish configuration.

7.4.1 Starting Web Front End

To start the web front end, change directory to the root of the source code base, and run `./script/server`. This will start the server in the foreground in its development environment. There are a number of daemons that will run Rails based web applications. Some include, Mongrel, Thin, and Passenger. Zackup's web front-end has been tested with Thin, but all should work. After running the command output similar to figure 7.5 should be seen. If there are any errors and the server refuses to start ensure all the dependancies listed in previous sections of this chapter are installed.

```
$ ./script/server
=> Booting WEBrick
=> Rails 2.3.3 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2009-09-26 22:12:07] INFO  WEBrick 1.3.1
[2009-09-26 22:12:07] INFO  ruby 1.9.1 (2009-07-16) [i386-darwin10]
[2009-09-26 22:12:08] INFO  WEBrick::HTTPServer#start: pid=49927 port=3000
```

Figure 7.5: Output of Web Front-End On Start

7.4.2 Create First User

1. Navigate to the web front-end. Typically this is located on port 3000 of the server. You will be directed to a interface similar to figure 7.6 asking for login credentials.

2. Since the service does not have any users yet, click on the register tab, to create a new user.

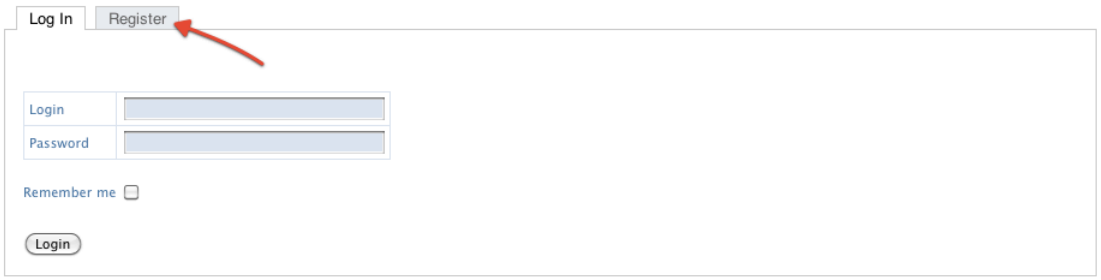

Figure 7.6: User Login Interface

3. Now at the registration page similar to figure 7.7, fill in the requested information to make a new user.

Figure 7.7: User Registration Interface

4. Now the first sight the web front-end, a page that gives various stats on the current user. This interface looks like figure 7.8.

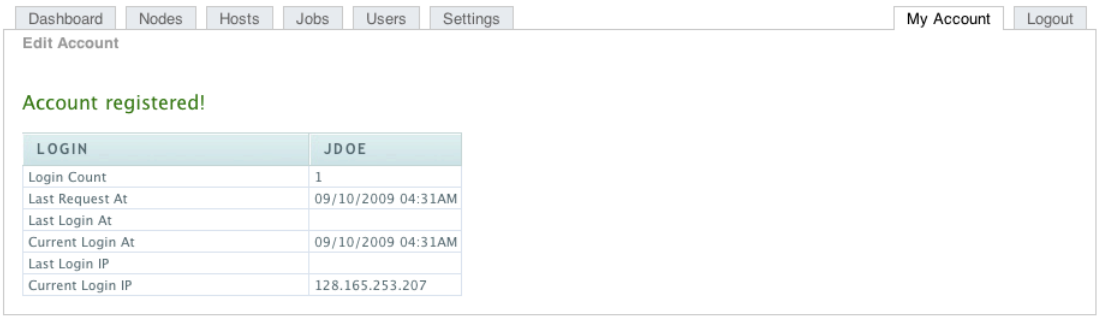

Figure 7.8: My Account Interface

The first user is now successfully created for the Zackup service. It is recommended that unless self-registration is desired that the **Registration**

preference is turned off under the Settings tab.

7.4.3 Adding Nodes

The next task is to add a node to the web interface. A node is either a scheduling daemon or backup daemon. Currently only the Backup Daemon checks to ensure it's a node in the database before continuing the start up.

1. Navigate to the Node interface by clicking the **Node** tab.

2. Seen currently is **No Records** as no node has been created.

3. Click the **New Node** link as seen in figure 7.9.

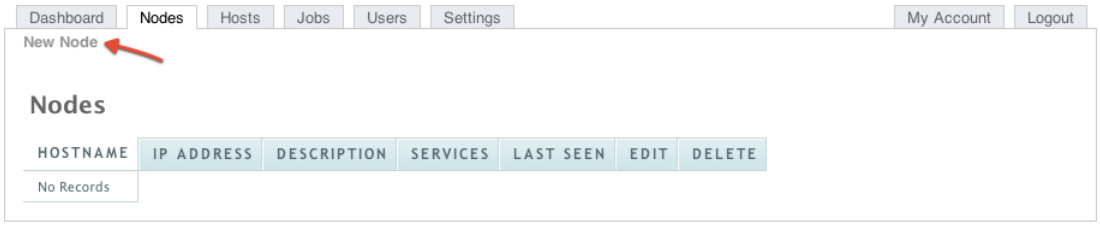

Figure 7.9: Node Index Interface

4. Fill out the presented fields on the Create New Node page. These attributes should match the ones set in the Backup Daemon's setting.yml file. In this example all entities of the Zackup service are going to run on the same server.

5. Ensure both Backup Node and Scheduler Node is checked as shown in figure 7.10.

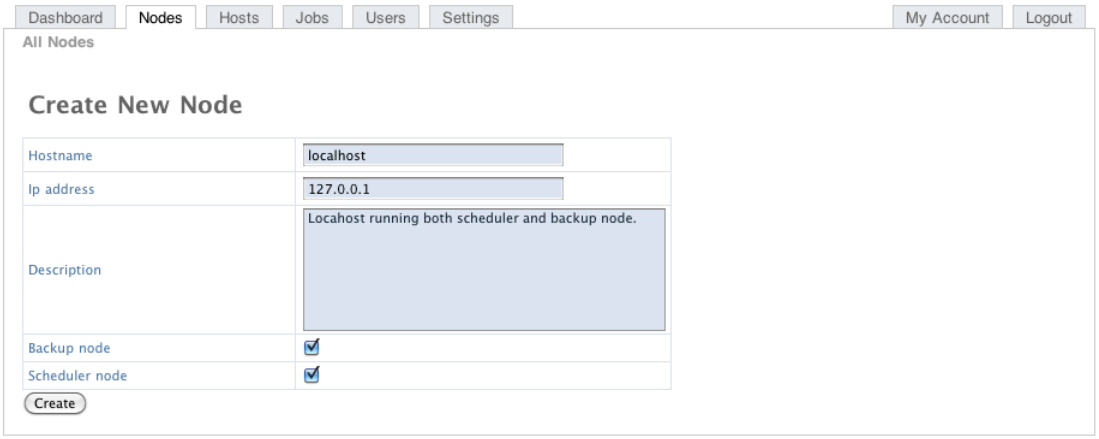

Figure 7.10: Create New Node Interface

6. After the node is successfully added it will be displayed in a similar fashion to figure 7.11. Note the **Last Seen** field. This field is updated by the Backup Daemon each time it iterates. This field can be an easy way to check if the Backup Daemon is still running.

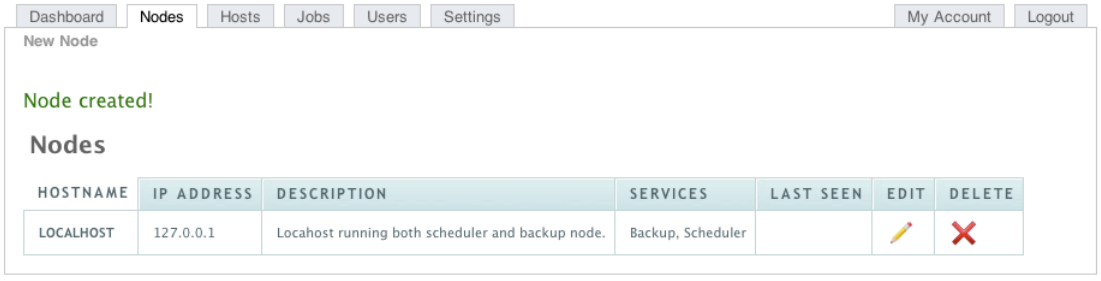

Figure 7.11: Node Index with New Node Showing

At this point the Backup Daemon can be started by following the instructions in section 7.3.4.

7.4.4 Adding Hosts

Now the Zackup service is ready to have client's added to it for backup.

1. Start by going to the **Host** tab.

2. As with the Nodes interface initially, **No Records** is displayed as no hosts have been added.

3. Click on the **New Host** link as indicated in figure 7.12.

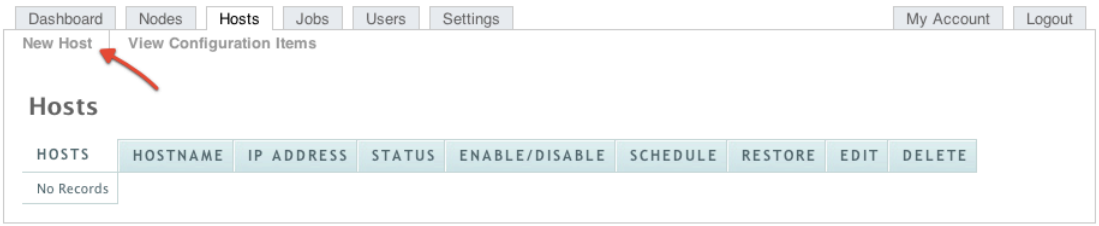

Figure 7.12: Empty Host Index

4. Enter in information for each field, ensure the information is valid for the client.

   Note: Quota is specified in the following format, 1G, 1M, 1K, 1B for 1 Gigabyte, 1 Megabyte, 1 Kilobyte, and 1 Byte respectively.

Note: Zackup will try to resolve the given hostname. If it cannot, it will use the IP address.

5. Now select **Host Type**

6. A select box will offer SSH and Samba as choices as shown in figure 7.13. These correspond to the method of file transport to be used for the client.

   Note: Currently only SSH is implemented.

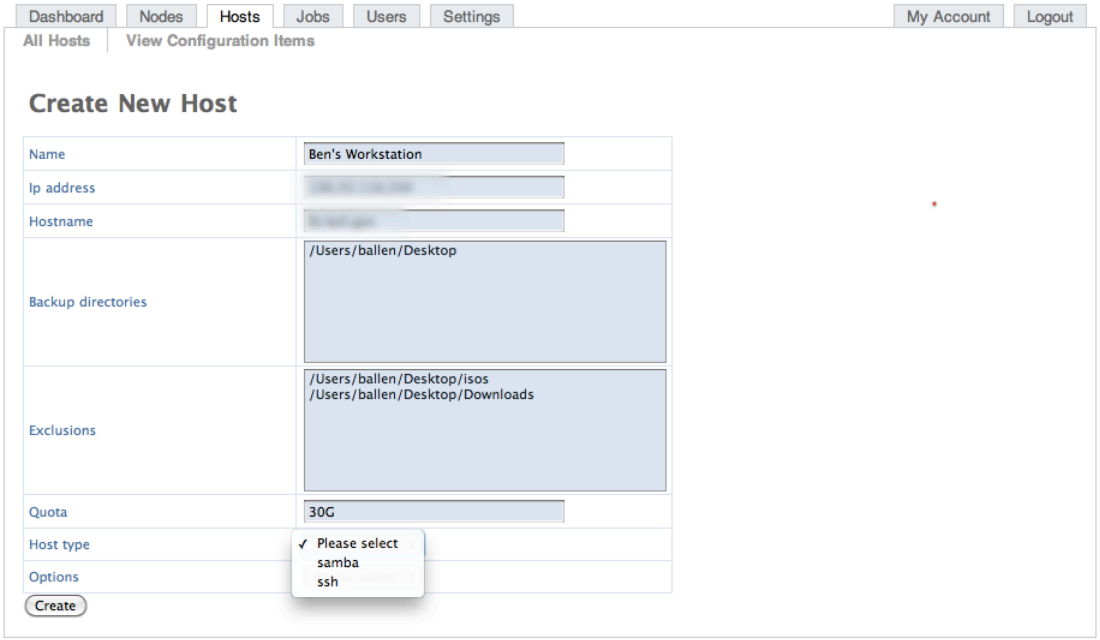

Figure 7.13: Creating New Host and Selecting Host Type

7. When you select a Host Type, additional fields will be added to the

form. Continue filling out the new fields. An example of an SSH form filled out is shown in figure 7.14

Note: The Options select field currently does not contain any additional options, but was implemented for future use.

Note: If the entered SSH private key is not encrypted, i.e. was created without a password, just leave **Ssh Private Key Password** blank.

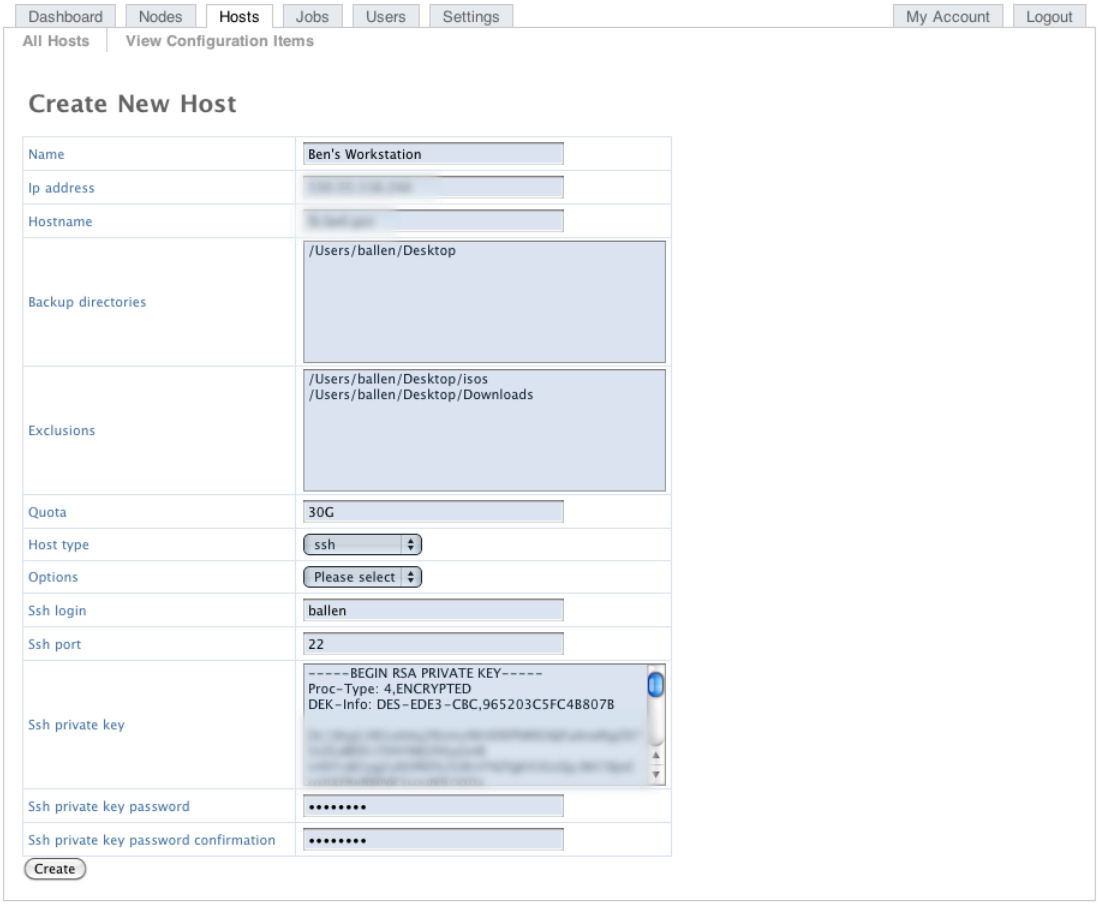

Figure 7.14: Creating New Host with SSH Host Type Select and Form Filled

8. Finish creating a new host by pressing the **Create** button.

At this point a new host is created, a host still needs at least one schedule before any files will be backed up.

7.4.5 Adding Schedules to a Host

A schedule is the method Zackup uses to determine when backups should be run for a host. While schedules are specific to a host, snapshots and all other backup related activities are specific to a schedule. To setup a schedule for the host previously created in this guide continue on with the following instructions.

1. Navigate to the **Host** tab.

2. Select the **Schedule** icon of the host as illustrated in 7.15.

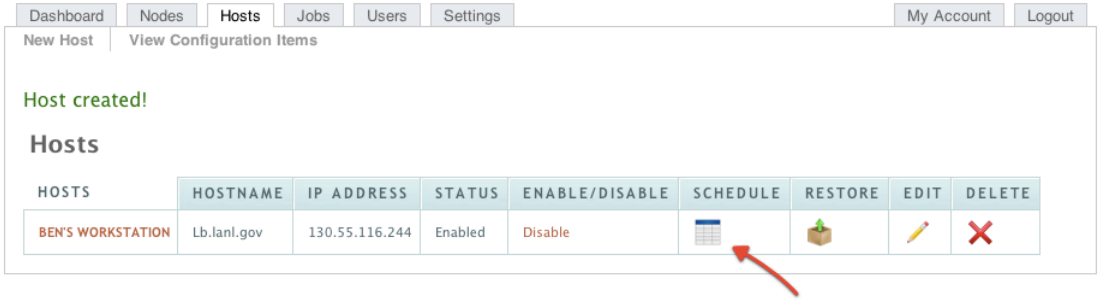

Figure 7.15: Host index with New Host Showing

3. The schedules index for the select host will be shown. As this is a new

host **No Records** will be shown as seen in 7.16. Click on the **New Schedule** link.

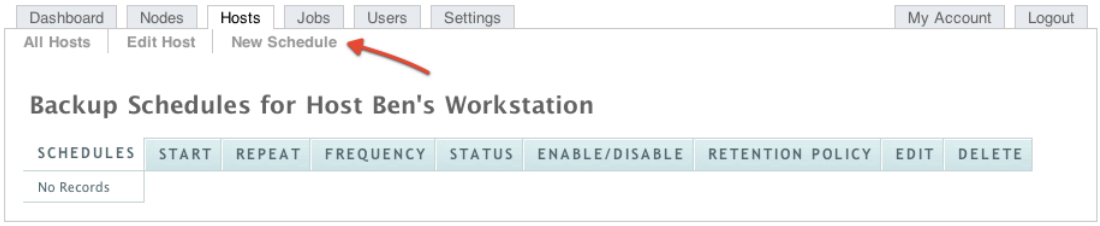

Figure 7.16: Schedule index with No Records

4. The interface for create a new schedule is not presented, as seen in 7.17.

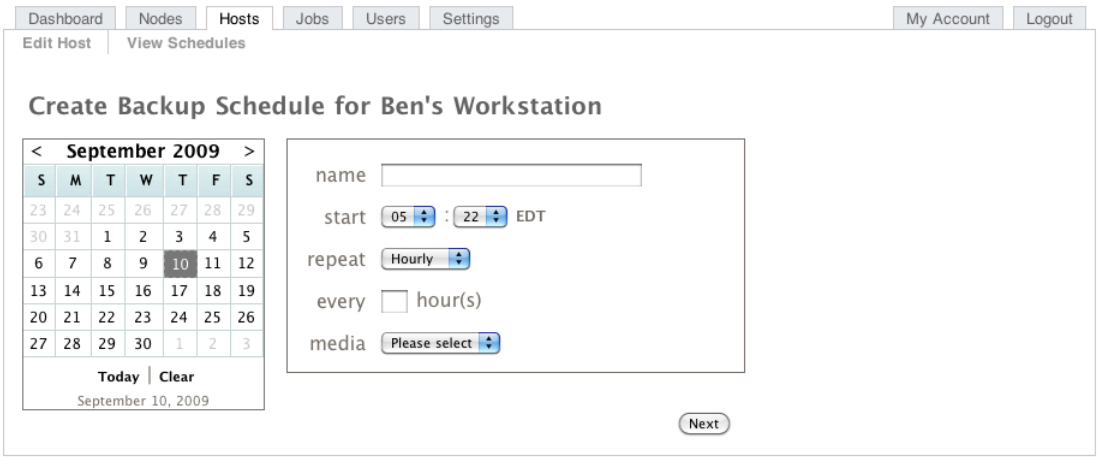

Figure 7.17: Schedule Creation Interface

5. There are four different types of schedules. Hourly, Daily, Weekly, and

Monthly. Each are shown in figures 7.18, 7.19, 7.20, and 7.21 respectively.

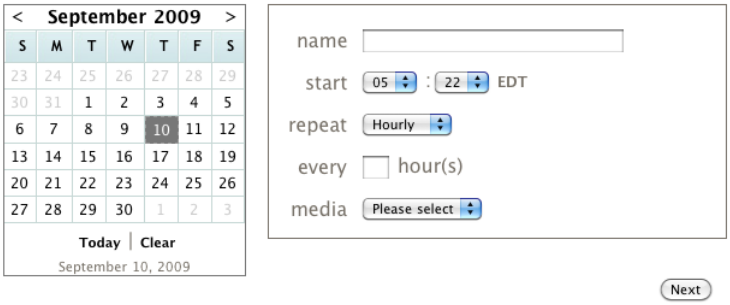

Figure 7.18: Schedule Creation Interface Showing Hourly Schedule

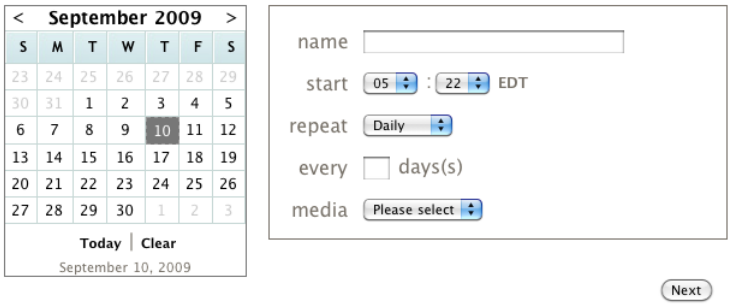

Figure 7.19: Schedule Creation Interface Showing Daily Schedule

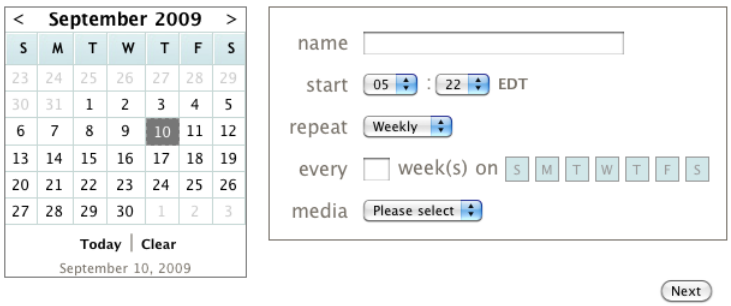

Figure 7.20: Schedule Creation Interface Showing Weekly Schedule

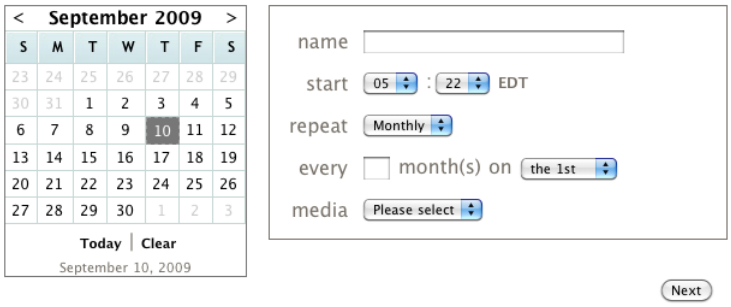

Figure 7.21: Schedule Creation Interface Showing Monthly Schedule

6. Select the type of schedule required, and fill in the rest of the fields.

   Note: The calendar on the left of the page is to set the date of the start at date and time of the schedule. This can be used to post-pone the schedule from starting for a certain amount of time.

7. Select the appropriate media. Media corresponds to any node with the backup service checked, and as such corresponds to specific Backup Daemons.

8. Press **Next** when finished.

9. The retention policy page is now shown, refer to figure 7.22.

   Note: A schedule is not complete without a corresponding retention policy. As such any schedule that does not have a retention policy will be skipped by the Scheduling Daemon.

10. Fill out the desired attributes of the retention policy

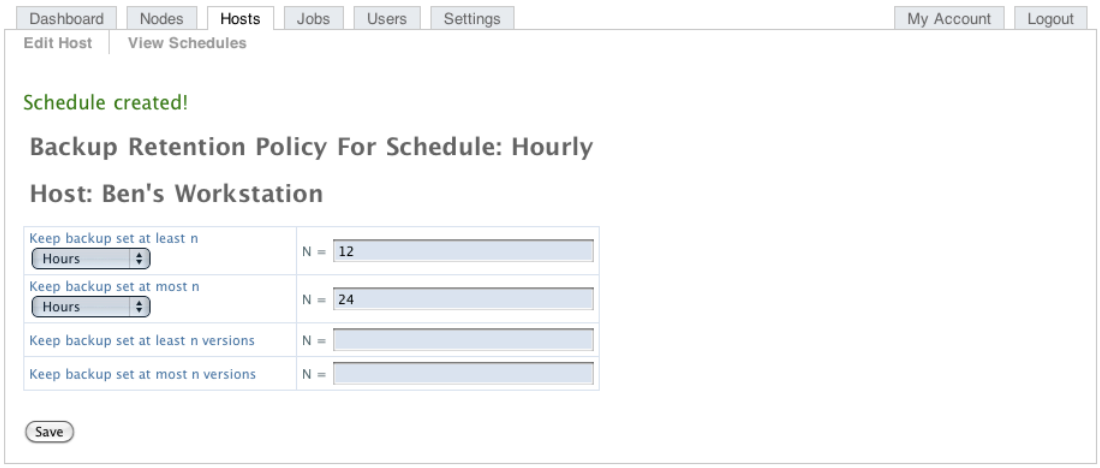

Figure 7.22: Retention Policy Creation Interface

11. Press the **Save** button when finished.

7.5 Summary

At this point the newly added host will be backed up on the schedule just specified. Backup sets will be expired based on the retention policy specified for the schedule. the rest of the web front-end is self explanatory and should not require the same detailed directions as seen in this chapter. Just remember this release is Zackup's first release and there are bound to be bugs, lacking features, and overall work flow issues that will be resolved in future released. If any bugs or feature requests are discovered please feel free to report them at http://github.org/unilogic/zackup/issues.

8

Limitations

8.1 Introduction

During the design and implementation of Zackup a number of limitations of the software was found. In addition to software limitations, some limitations, where noted, were self imposed to limit the scope of the Zackup project.

8.2 Programatic

Maintaining file metadata such as permissions, and ownership after being copied from the user is not possible in many cases. Due to the fact that Zackup can copy data from many diverse filesystems to ZFS, certain unknown loss in metadata likely occurs. Solving this limitation was not a main concern

of this project. However, the data within files is transferred correctly and will maintain its integrity. The file transferring code of this project is modular in design allowing future work in this area.

FreeBSD's ZFS implementation has a path limit of 88 characters to access filesystem, snapshot folder, and subsequent subdirectories. ZFS snapshots are accessed from the filesystem itself from a **.zfs** folder in the root of the filesystem. Note, this is an option and must be enabled. As a result of having a limited path length, less random names for each filesystem had to be used. There is a potential, although small, that the setup job will fail if an existing filesystem with the same name of the randomly named new filesystem exists. As a precaution the Backup Daemon checks for existing filesystem before creating a new one. If an existing filesystem is found, the Backup Daemon loops up to five times, an arbitrarily picked number of times, to find a new filesystem name. Filesystem names consist of the host's IP Address followed by a 5 character lower and upper case alpha-numeric random string. The IP address was used as it implicitly has a guaranteed maximum string length.

Symbolic links in a restore set do not resolve if they are absolute paths and/or if the symbolic link's destination is not included in the backup. As a result of another limitation of the Ruby Minitar library, symbolic links that

do not resolve are explicitly ignored and not included in the restore. The Minitar library currently does not know how to meaningfully handle symbolic link filetypes, and as a result sees it as a normal file. Thus it currently tries to copy the file it references and has no option or ability to copy the symbolic link itself. No other pure Ruby library that implemented tar archiving was found.

8.3 Security

Authentication and other security matters was not a primary focus of this project. Related limitations to this area is the authentication of the user to the web back-end. Currently any user authenticated or not with enough guesses will be able to download existing restore sets. As a precaution restore set filenames consists of sixty-four random characters. Ideally the web back-end would check the user has proper authentication to access a given file. This limitation was a self-imposed limitation set during the design stage of Zackup to limit the project's scope. See the Future Work chapter for more ideas regarding this limitation.

8.4 Policy Enforcement

Ensuring stored data is legitimate in nature and does not violate any given policy other than simple storage quotas is not a current feature of Zackup.

No virus or malware detection is built into this project. However, ZFS does have the ability integrate third party virus scanning software. Potentially the output of that software could be tied back into the web front-end. This limitation was also self-imposed and limited due to project scope.

8.5 Summary

The programatic, security, and policy enforcement limitations described here were a mix of unexpected unresolvable issues found during development as well as a number of self-imposed limitations to limit the scope of the project. Likely with sufficient work and additional outside help all the above limitations could be solved. Look forward into the future work chapter for some suggested solutions and workarounds for these limitations.

9

Future Work

9.1 Introduction

The future work chapter describes many of the authors wishes and ideas for new versions of the Zackup program. Many items were not implemented in the current version of Zackup due to the feature's complexity, unforeseen design issues, or simply wishful thinking from the author during and after design or implementation.

9.2 Security Features

9.2.1 Symbolic Link Information Disclosure

Ensure symbolic links that resolve outside of the user's ZFS filesystem are ignored. Currently when restoration jobs are executed, Ruby's MiniTar library resolves all symbolic links to their destination files. If a symbolic link was created to point a system or otherwise sensitive file, the restore job would happily copy that (as long as it had access to it) into the restore tar archive. The tar archive would be made available for download to the user, and the user would now have a copy of the sensitive file. This could also be used to restore other user's files without permission. To resolve this issue an additional check will need to be made when iterating through snapshots to create the restore set. The check will ensure if object is a symbolic link, and if so that that symbolic link does not resolve outside of the given user's filesystem. If it does, which often it might in the case of symbolic links using absolute paths, the path to the user's filesystem will be prepended to the path. If that path resolves to a file it will be added to the restore set. If it does not resolve the symbolic link will be ignored. Ideally when symbolic links are ignored, it would be logged to the database, as well as recorded in a text file included in the Tar archive.

9.2.2 Web Front-end Role Backed Authentication

Web site administrative and user roles need to be implemented. Currently all users that have access to the web front-end are considered admin users. In other words, all users are allowed to carry out all functions available on the website. Ideally a number of roles would be setup, and functions of the website added to the roles. Users would then be assigned to given roles depending on their needs. In addition, an interface that allowed custom roles to be created based on individual functions of the web front-end should be created. This would allow for greater flexibility for unseen users and administrative uses of the service.

9.2.3 Web Back-end Authentication

Restore sets are available to the public if the filename of the restore set file is known. Currently a long random, 64+ characters, filename is used for the Tar archive. This approach is utilizing security by obscurity. Making the filename complex is not a substitute for real authentication. However, to properly authenticate users to the web back-end, the back-end server will have to support some method of using the central database. In addition, this web server will have to effectively determine if a user is authorized to download a specific restore.

The best way of accomplishing the previous task will likely be to create a barebones Ruby on Rails application whose sole purpose is to check authentication, or to ask for authentication and then serve up the requested restore set file. This should be possible by using the same authentication library as the web front-end, sharing the same cookie information, and authenticating against the same central database. Ideally the user could log into the web front-end, and click on a restore set's URL and never be asked to enter additional credentials. If a user was to go directly to the restore set's URL, he or she should be redirected back to the web front-end to be authenticated.

Web Back-end Authentication was specifically stated to be out of scope of this project. It adds an entirely new entity into the Zackup service, and was identified early on as being future work for this version of Zackup.

9.3 Functionality Features

9.3.1 Quotas

Currently quotas are set on a per ZFS filesystem basis. Since a new filesystem is created for each schedule. A host's quota is not accurately enforced. A user can simply create a new schedule and have double the storage space allocated to them. This is a relatively hard problem to solve with Zackup's architecture. As a filesystem's quota is set at creation, if a new schedule was

to be created existing filesystems' quotas would have to be altered to redistribute the user's total quota. A more elegant solution, at least from a coding aspect, would allow the user to piece meal their quota on schedule creation. In other words, allow the user to select a percentage or portion of their total quota on schedule creation. On each subsequent schedule creation the user would only be allowed to allocate the remaining portion of their quota. If all portions of their quota were allocated, no new schedules would be allowed to be created, and as a result no new filesystems would be created for a user. The user would only be allowed to use their total allotment of file space thus solving the noted limitation.

9.3.2 Backup Daemon and Threading

As a matter of simplicity the Backup Daemon as well as the Scheduling Daemon are programed to function in a serial matter. For example, the backup daemon starts by fetching data from the database, parsing that data, running the first job, running the second job, and so on. One operation takes place at any given time. Where this becomes a problem is long running jobs. If a long backup job needs to run, it delays any other assigned jobs waiting to be run. However, a serial execution design like this is less complex than the alternatives of running a threaded daemon or running a daemon with

child processes. Concurrency becomes an issue once its decided to have multiple entities potentially access the same bit of data or other resources. In the Backup Daemon's case, additional code would need to be implemented to ensure an about to be created thread or child process will not work on the same client filesystem. In addition, a rate throttler would need to be implemented so only a certain number of operations would be running concurrently. This would ensure the system's network bandwidth, memory, disk IO, and CPU resources were not exhausted. The obvious benefit of a design that allowed concurrent operations would be speed increase, as long as the system resources were not over consumed by those additional operations. In addition to a speed increase, the chance of running jobs on-time would increase dramatically. These features were not implemented because of their inherent increase of complexity, and as such are relegated to future work on the project.

9.4 Usability Features

9.4.1 Stats and Backup Daemon System Health Reporting

The Backup Daemon needs to report back additional stats on the daemon's environment. Currently host CPU load average and backup storage volume used and available disk space are tracked. In addition, each schedule's indi-

vidual ZFS filesystem available and used space is tracked. Additional system statistics like free memory, performance of the disks, i.e. The ZVol's read and write IO, as well as stats on any applicable caches. Potentially also useful would be integrating in SMART disk status, so an administrator would get a notification if any of the system's hard drives had a SMART error. Overall these items were deems not critical to day to day tasks of Zackup and were not included in its first version. In addition, there is a fine line between storing too much statistical information into the database, and not enough. If more data is stored than the typical administrator needs, the database takes on extra load, as well as uses up additional disk space. On the flip side, if there is not enough data for the typical administrator, potentially serious errors or error conditions could go unnoticed. With the basic data Zackup provides the user and administrator has the essential data displayed to him or her.

9.4.2 Load Balancing Backup Daemons

Scheduling Daemon or web front-end interface ideally should automatically pick the best storage daemon for a schedule. Currently when a schedule is created the administrator or user must manually select which backup daemon node to use. This implies the user knows which node has enough storage

space, is the least busy, as well as is fastest network wise server available. Available storage space, and CPU load is graphically shown in the node's show view in the web front-end interface. However, deriving which node is closest to the client is more difficult and requires outside knowledge of the network, and such is not a easy metric to program for automatically choosing the best storage daemon. Metrics like ping response time, or integrating a small test download from each backup node are two ideas that would have merit in this area. The manual method implemented in the current version of Zackup has the lowest overhead as far as complexity of code, and complexity of network requirements. As a result, to stick to one of the projects main goals of having a small code base, the manual load balance method was chosen for the first release of Zackup.

9.4.3 File Transport Features

The backup daemon only supports using RSYNC over SSH as a file transport method. Initially, Samba and FTP were anticipated to be supported in Zackup's initial release. Both protocols have inherent problems that prohibited them from being used in Zackup's design.

FTP does not allow for differential backups. In other words, a whole file needs to be downloaded from the client any time its updated. Even if the

update is only one block in size (typically 512 bytes). This results in a huge waste of network resources and time. RSYNC over SSH solves this problem by running the RSYNC binary on the client as well as the server. The RSYNC binary on the client indexes new or updated files, it then transfers only the changes in the file. In the above example if one block of data was altered only that block worth of data would be transferred, where the server side copy of the file would be updated with the changes.

Samba runs into a similar issue as FTP. The author's original idea was to mount the Samba share on the server. Then use RSYNC only on the server-side to do differential copies. The first issue is RSYNC cannot reliably work as it relies on the file metadata to determine if the file is updated. The file metadata, mainly the timestamps are not reliably translated into Unix's MAC times (in this case FreeBSD) from Windows. The only option in RSYNC left to reliably make sure you catch all changed files is a checksum option. However this would require RSYNC to read the entire file across the network to checksum it. Thus any differential backup ability of RSYNC has been negated by the need to do checksumming on the file server-side.

As a result of not being able to effectively implement the above two file transport protocols, the Windows client platform is currently largely un-

supported. A workaround includes installing a Windows version of RSYNC and SSH. The most common method of doing this is to use the Cygwin http://www.cygwin.com project. While this allows the client to run the needed software, this install method is by no means user friendly. It is also cumbersome to keep up to date. As a result, a new method of file transport must be found or created. Most commercial backup solutions offer a custom built backup client, i.e. IBM's Tivoli, EMC's Retrospective, Backblaze, Mozy, JungleDisk, etc. Developing a custom backup client that would run on the Windows platform is well beyond the scope of this project as well as the scope of the author's ability. However for future work, this possibility must be considered if the Windows platform is to be fully supported as a client.

An additional file transport method using ZFS Send and ZFS Receive over an SSH connection should be implemented. Adding this file transport method, i.e. ZetaBack, would allow a few interesting possibilities. First, Backup Daemons would be able to mirror each other's filesystems, or a tertiary Backup Daemon could keep second copies of data stored in ZFS filesystems that directly store client's data. This would allow an additional level of redundancy in the backup. In additional to increased redundancy, the supported clients would increase. Solaris or FreeBSD based clients running

the ZFS filesystem could be backed up with this facility instead of RSYNC over SSH. This method of transport was not implemented as it was outside the scope of this project.

9.4.4 RSYNC Options

Ignore RSYNC return code 23, partial download during a backup job, and treat this return code as a successful synchronization from the client. This return code is typically the result of a permission issues on the client not allowing the configured user to access a subdirectory of the configured backup directory. RSYNC reports this as an error as it was not allowed to copy all files and directories. Some clients may want this error skipped. An option when setting up either the host or schedule should be added to tell the RSYNC process to ignore this specific error. The author feels this return code should actually always be treated as an error. However, other similar backup software, mainly BackupPC, has this option and may be desirable to have in the future.

As a result of one of the limitations of the Minitar Ruby library of not being able to properly deal with symbolic link file types, it would be a useful work around to force RSYNC to download the referenced files of symbolic links instead of the symbolic link itself. In addition, this option would insure

the referenced files were backed up and available for restore. If for example a symbolic link pointed outside of the configured backup directories, currently the symbolic link would be backed up, and its referenced file would not. This results in a broken symbolic link for the Backup Daemon which is ignored and not included in restore sets.

9.4.5 Web Interface Improvements

A dashboard should be added to give the user or administrator a quick overview of the status of the server, their backup sets, current running or erred jobs, as well as any other quick glance type information. The dashboard will be displayed to the user on login and as a result will be the first view the user or administrator has of the backup service.

9.4.6 Error Reporting

Better highlighting of errors should be created. A user currently has to look at the job's show view to find out what went wrong. Jobs that have errors should be shown predominately in the mentioned dashboard. This view will allow the user to quickly evaluate errors, and expeditiously fix a given problem.

RSYNC often gives cryptic errors related to permission problems, SSH key authentication failing, and more. These common RSYNC errors should be

translated for the user to give a more meaningful error.

For example **Partial transfer due to error** and **Partial transfer due to vanished source files** are common errors. In all likely hood the first error could be translated for the user to **Partial transfer due to permission denied on subdirectory**. The second error could be translated for the user to **Partial transfer due to files being moved or deleted while Zackup was doing a backup**.

9.4.7 Server Platform

All platforms that support ZFS should be thoroughly tested for support for Zackup. Bugs or incompatibilities will need to be identified and corrected. Current known stable ZFS platforms are all the various current Solaris operating systems and FreeBSD 8.

9.5 Summary

As seen, Zackup still has a lot of work to be done. As with any large programming project, one can almost infinitely add additional features. It comes down to amount of required time, demand from users or other stakeholders, complexity of the feature, and finally the amount of burden it adds to the program as far as performance and maintainability.

10

Conclusion

The developed software meets its initial goals and allows for a scalable centralized backup service that takes advantage of the ZFS filesystem for a great number of the common backup set tasks. With the use of already available Ruby libraries, the custom written code was kept to a minimum without sacrificing the quality of the product.

The secondary objective of this project was to implement a proper backup set retention plan that will most efficiently take advantage of ZFS' capabilities. With the options of using the following strategies, "keep at least N days", "keep no more than N versions of the file", "after N days delete the file", and "lastly do not delete before N days have passed" it was found that

the last three strategies were not feasible with ZFS. This is because ZFS snapshots were used, and a snapshot is taken of the entire filesystem, not individual files. As a result, any strategy that relied on operating on a single file was not possible. This did lower the granularity of control over the retention of files. However, it does not appear to add a severe additional burden on storage resources. The implemented retention policies methods were: "keep at least N versions", "keep no more than N versions", "after N time units expire", and lastly "do not expire before N time units have passed".

The last objective of this project was design and implement this backup service in a way that was scalable. With the implementation of the message queueing architecture for communication of jobs between all entities of the service, this goal was accomplished. Multiple backup daemons can be used to distribute the load of users vertically across additional servers. While the current implemented methods of load balancing are limited to being manual, future work should prove to allow automatic balancing of Backup Daemons.

Overall, the Zackup service has proved to be a stable and useful software. It has been seen to run a minimum of a week without interruption or error. As such is should be considered as successful first version. As with all first versions of software, additional features are still needed, and bugs are going

to be found. With the open source community's involvement in the project, hopefully Zackup will prove to useful piece of software to the information technology industry.

Bibliography

[1] Ted Bonkenburg, Dejan Diklic, Benjamin Reed, Mark Smith, Michael Vanover, Steve Welch, and Roger Williams. Lifeboat: An autonomic backup and restore solution. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 159–170, Berkeley, CA, USA, 2004. USENIX Association.

[2] Jeff Bonwick. Zfs: The last word in filesystems [online]. Available from: http://blogs.sun.com/bonwick/entry/zfs_the_last_word_in [cited September 25, 2009].

[3] Jordan Hubbard and Satoshi Asami. About the freebsd project [online]. Available from: http://www.freebsd.org/doc/en/books/handbook/history.html [cited September 25, 2009].

[4] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *OSDI '99: Proceedings of the third symposium on*

Operating systems design and implementation, pages 239–249, Berkeley, CA, USA, 1999. USENIX Association.

[5] Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving internet catastrophes. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2005. USENIX Association.

[6] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative internet backup scheme. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[7] OmniTI. Zetaback zfs backup and recovery tool [online]. Available from: https://labs.omniti.com/trac/zetaback [cited September 24, 2009].

[8] Inc. Sun Microsystems. Zfs faq at opensolaris.org [online]. Available from: http://opensolaris.org/os/community/zfs/faq/#whatstandfor [cited September 25, 2009].

A

Appendix: Key Terms

The following terms are used through out this proposal and are assigned a specific meaning as noted below.

- Backup Service - A entirety of the developed software, i.e. the main subject of this thesis.
- User - A person who wishes or requires their data to be backed up by this backup system.
- Administrator - A user with granted permissions to add, update, and delete all configuration information from the web front-end.
- Job Queueing - Storing jobs in a database table where both the creator or the job and executor of the job can access and update the job's data.
- Jobs - A task to executed by any software entity in Zackup. Common tasks are pulling a backup set from the client, cleanup old backup sets,

and creating a restore set.

- Backup Sets - A set of files specified by the user to be backed up. A single backup set typically has a specific time and date associated with it.
- Restore Sets - A set of files specified by the user to be restored and made available to the user for download.
- Scheduling Daemon - Daemonized software responsible for running jobs at the configured time.
- Schedule - A configured time that allows the Scheduling Daemon to determine when to create backup jobs.
- Backup Daemon - Daemonized software responsible for pulling files from users, storing users' files, creating restore sets, and cleaning up unneeded files.
- Web Front-end - A HTTP based interface for the user to configure the backup service, retrieve storage pool statistics, initiate manual backup jobs, browse stored backup sets, and initiate restore jobs.
- Web Back-end - A HTTP service whose sole purpose is to serve created restore sets to users.
- Storage Pool - The total resource set of storage configured to be used for the backup service.